IBM Rational University

Rational. software

# Pattern Implementation Workshop with IBM Rational Software Architect

*RD801/DEV498 April 2007*
*Student Manual Volume 2*
*Part No. 800-027312-000*

IBM Corporation
Rational University
Pattern Implementation Workshop with IBM Rational Software Architect
Student Manual Volume 2

April 2007



Printed in the United States of America.

This manual prepared by:
IBM Rational Software
555 Bailey Ave.
Santa Teresa Lab
San Jose CA 95141-1003
USA

# Contents

## Module 18: Packaging Artifacts



## Module 19: Summary and Conclusion



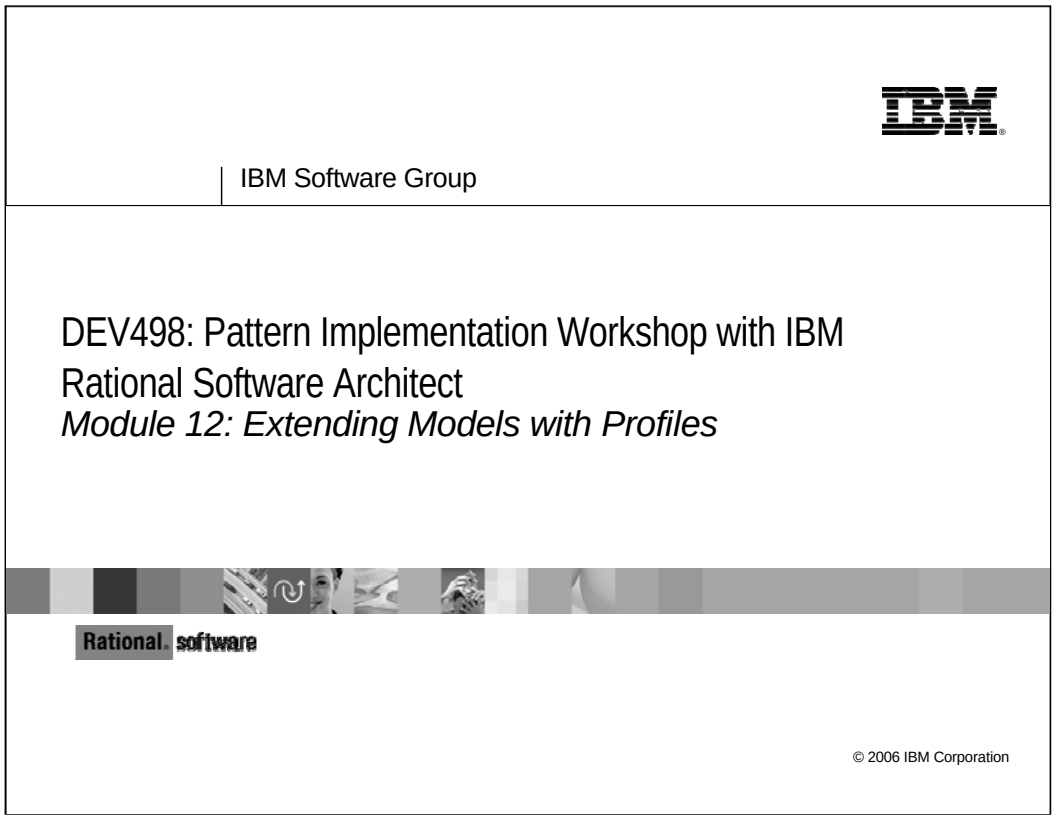
## Module 20: Advanced Transformation Topics



## Module 21: Introduction to GMF



## Module 22: XPath: XML Path Language

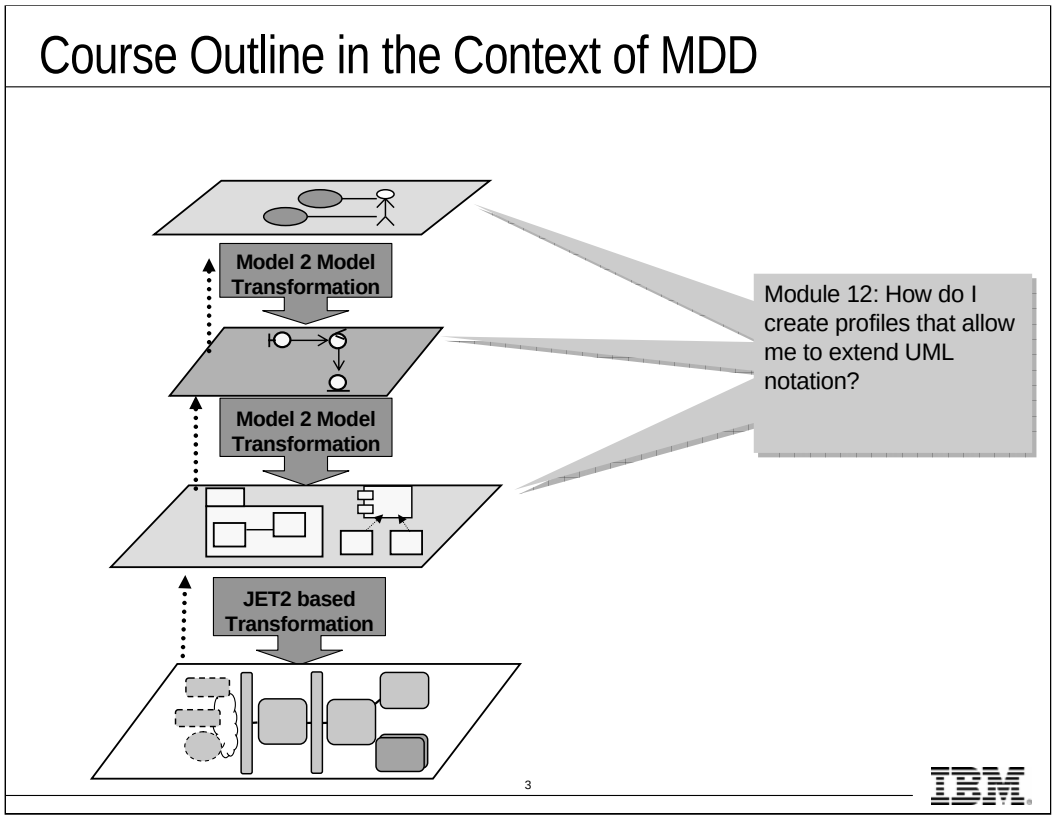



## Contents

# Creating UML Profiles

- Objectives:
  - Describe UML profiles
    - The uses for profiles
    - How to design a profile
    - How to create a profile in Rational Software Architect
    - How to customize profiles by adding icons and constraints
    - How to apply a profile to a model
  - Describe the relationship between profiles, models, UML patterns and transformations.
  - Create a UML profile in Rational Software Architect

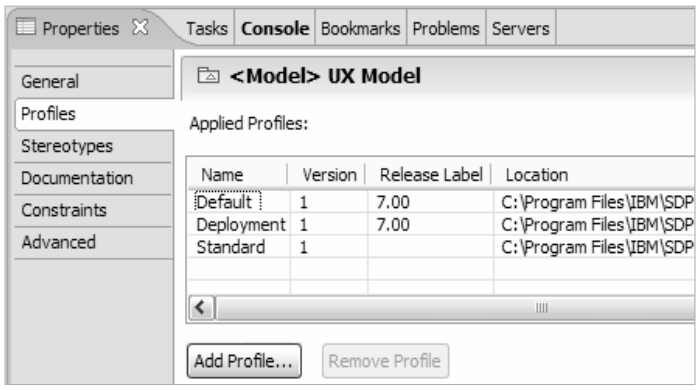
2




We will see this slide several times throughout the workshop. It will serve as a visual guide to the skills that you are learning, and to how they fit into MDD.

## What Are Profiles?

- Part of the UML standard
- Specializes UML for specific domains
  - Projects, process, industry, or technology
- Multiple profiles can be applied to a model




4

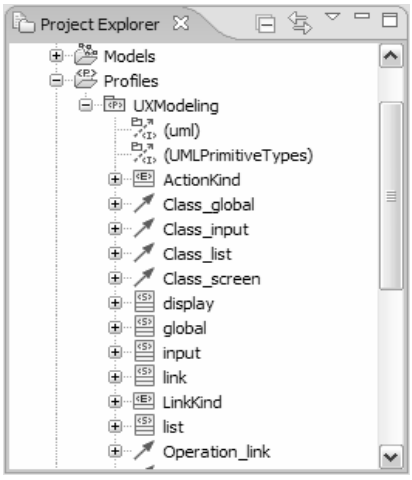
## Profiles (cont.)

- Profiles consist of stereotypes, which add semantics to model elements
  - Can be localized
  - Are versioned and formally deployed
  - Are stored in a single file: *<profile name>.epx*
- Stereotypes can include:
  - Properties
    - A name-value pair that captures additional information
    - Examples: package owner, class QA status, and addressed requirements
  - Constraints
    - Live and batch rules
    - Examples: *age > 18*, or *stereotyped class must inherit from library class*

5

## UML Profiles in Rational Software Architect

- Stored outside the model, using **.epx** files.
- Created using the Project Explorer view, in the Modeling perspective, and the Properties view.
- Existing profiles can be updated with new versions of the profile.
- Reference .epx files directly, or deploy via a plug-in and add the profile to Rational Software Architect.




6

IBM

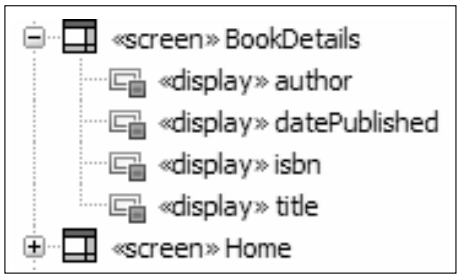
Rational Software Architect allows you to develop and apply UML profiles. You can use UML profiles to create model elements that reflect the semantics of a specific domain or platform. UML profiles are sets of stereotypes, tagged values, and constraints.

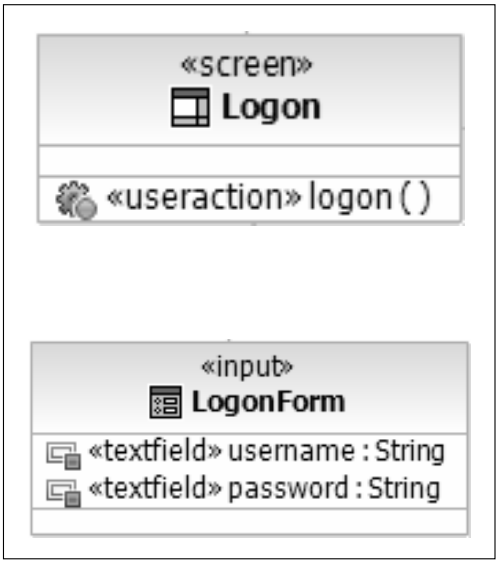
- **Stereotype:** Elements based on existing types or classes in the UML metamodel, that extend the metamodel. Stereotypes can extend the semantics, but not the structure of pre-existing types and classes.
- **Tagged Value:** A property as a name-value pair; the name is referred to as the “tag.”
- **Constraint:** A semantic condition or restriction.

Like the other extensibility features, profiles can be deployed as Eclipse plug-in projects. You author profiles using the Project Explorer view and the Properties view, rather than authoring them in the diagram editor. When the profile is complete, users can apply the profile to their models by selecting the target model in the Project Explorer view, and then adding the profile under the **Profiles** tab in the Properties view.

## What are Stereotypes?

- They add semantic meaning
- There are multiple stereotypes per element
- Their presentation includes
  - Project Explorer: text and icon
  - Diagram Editor: icon, shape, and text



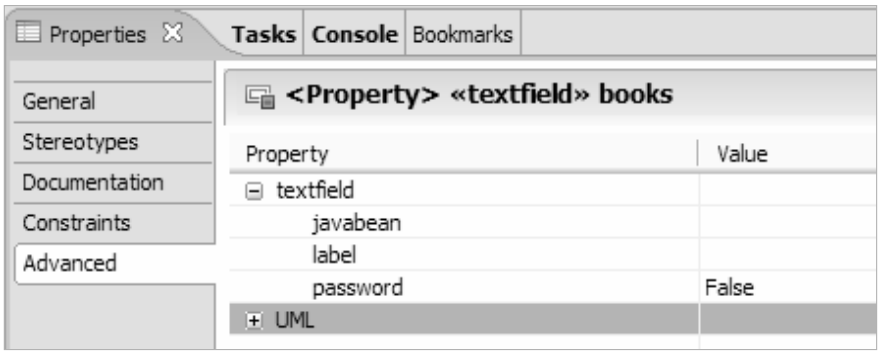




7

IBM

## What are Properties?

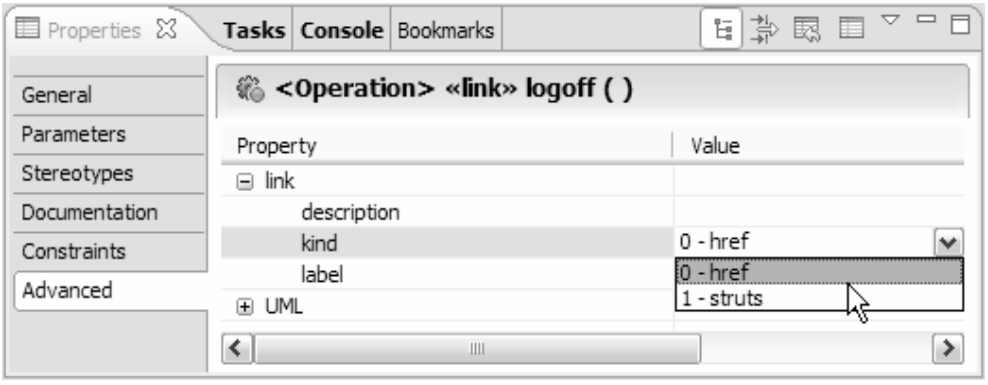
- Name-value pairs on model elements
- Called “tagged values” when applied
- Include types: String, Boolean, Integer, and Enum




8

## What are Properties? (cont.)

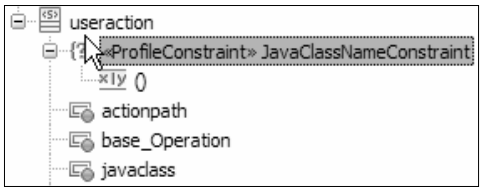
- Define your own enumerations
  - Restrict to predefined values



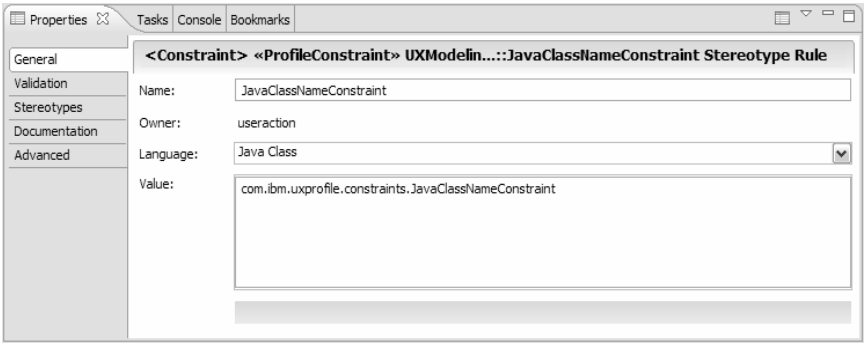

9

## What Are Constraints?

- Allow you to express a condition or restriction to which the element must conform
- Can be expressed in:
  - Natural languages or mathematical notation
  - Java
  - Object Constraint Language (OCL)




Project Explorer View




10

IBM

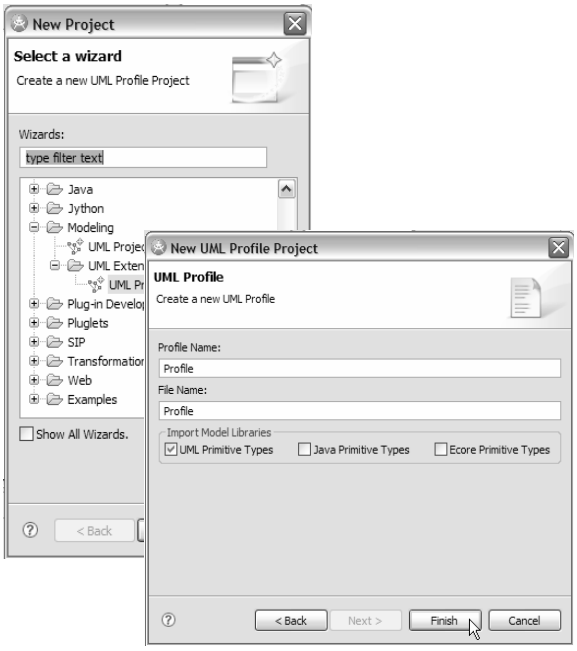
A constraint lets you refine the semantics of a UML model element by expressing a condition or a restriction to which the element must conform.
You can specify the language that you use to write the body of a constraint so that others who read the constraint can more easily understand its condition or restriction.
In Rational Software Architect, you can create constraints in the following languages:

- Natural languages such as English or mathematical notation (in UML comments)
- Java
- Object Constraint Language (OCL)

You can specify constraints within a stereotype in a custom UML profile. When you define Object Constraint Language (OCL) constraints, the constraints are validated syntactically. However, Java™ constraints are not validated syntactically. When you apply a stereotype to a model element, the attributes of the stereotype are added to the model element. Stereotype constraints apply to the attributes of the model element to which the stereotype is applied. The model validation process checks model element attributes for compliance with stereotype constraints. If you specify a value for the attribute that does not comply with the constraint, an error is displayed in the Problems view.

## Creating a Profile in Rational Software Architect

- Create a Profile Project
- Add stereotypes
  - Use extensions to connect the stereotypes to UML elements
- Add tagged values and constraints
- Test by applying the profile to a project
- Distribute as:
  - RAS file
  - Plug-in
  - .epx file




11

IBM

The key steps in creating a Profile in Rational Software Architect, are:

1. **Create a profile project:** Profile projects are a form of modeling project in Rational Software Architect.
2. **Add stereotypes, tagged values, and constraints:** These elements are added in the Project Explorer, using the Modeling Perspective, and modified using the properties view.
3. **Use extensions to connect the stereotypes to UML elements**: The Extensions page in the Properties view for the stereotypes allows you to apply the stereotype to specific elements (class, component, and so on) in the UML metamodel.
4. **Test by applying the profile to a project:** Verify that the profile is valid, and that it is semantically sound, based on the target domain or technology.
5. **Distribute:** Profiles are distributed as Eclipse plug-ins.

# Distributing Custom Profiles

- When ready for use, profiles have to be **released** and made available as an Eclipse plug-in
  - After release, modifications are restricted to adding stereotypes only
  - Profiles in plug-in form can be distributed as a RAS asset
- Tips:
  - Do not release the profile during development (perform release process for testing only)
  - Wait until the profile stabilizes before distributing it

12

IBM

## Demo: Create a Profile

The instructor will now show you how to:

- Create a Profile
- Add Stereotypes
- Specify Extensions
- Create an Enumeration

13

IBM

Watch your instructor create a simple profile.

## Tips for Creating Profiles

Use the following tips to guide your profile development:

- Look for published, existing profiles that may meet your needs
- Determine the level of abstraction that makes the most sense
- Identify the key terms in the domain that you are trying to represent in the profile
- Consider how profiles may work with custom patterns and transformations

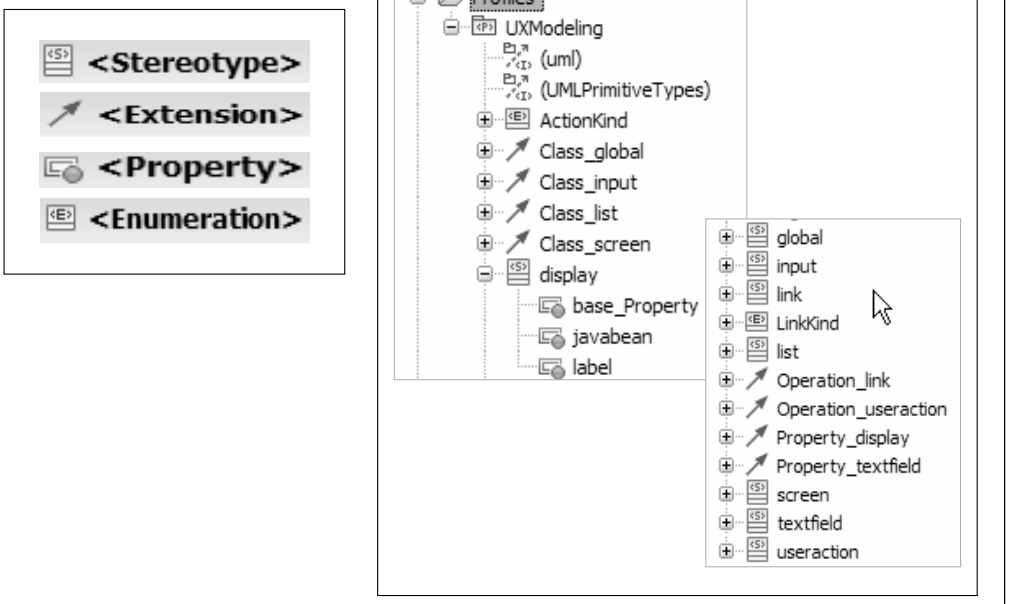
14

When a project calls for a new UML profile, look for existing profiles that may meet your needs before trying to design a new profile. Profiles may be available in internal RAS repositories or publicly available from IBM®'s developerWorks®, industry repositories, the OMG, and so on.

If it becomes clear that you must build a new UML profile, consider the following general suggestions:

- Determine the level of abstraction that makes the most sense for the types of models you will be creating with the profile.
- Identify the key terms in the domain that you are trying to represent in the profile. Note that the terms captured in the profile are not the elements of the solution, but elements used to describe the solution.
- Design the profile with the UML patterns and transformations in mind that will be used in the models to be developed with the profile.

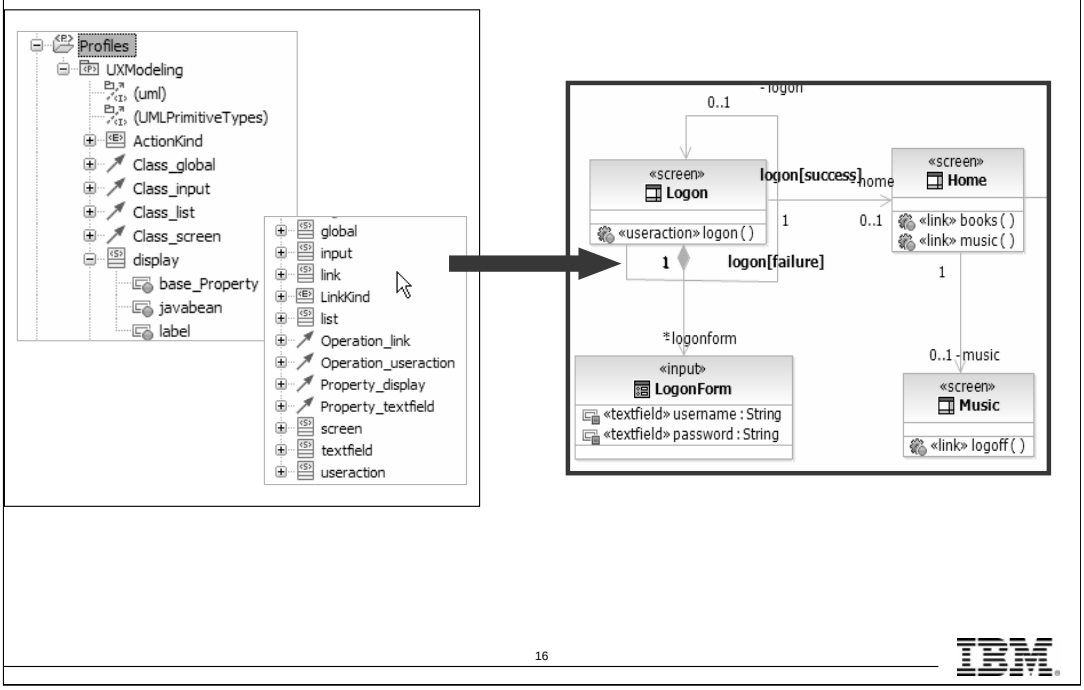
# UXModeling Profile




15

IBM

# Example UML Model with UXModeling Profile Applied




16

## Lab 9: Create the UX Modeling Profile

- Complete the following tasks:
  - Create the Workspace
  - Create the Profile
  - Apply the Profile to a Model
  - Add a constraint to the profile
  - Add the profile to a plug-in project

17

IBM

Complete Lab 9 in the student workbook.

# Review

- Describe the difference between a constraint and a property.
- How many profiles should be applied to a model?
- What languages can be used for specifying a constraint?

## Further Information

- Rational Software Architect Help Topics
- Web resources
- Literature


19

IBM

**Rational Software Architect Help Topics**

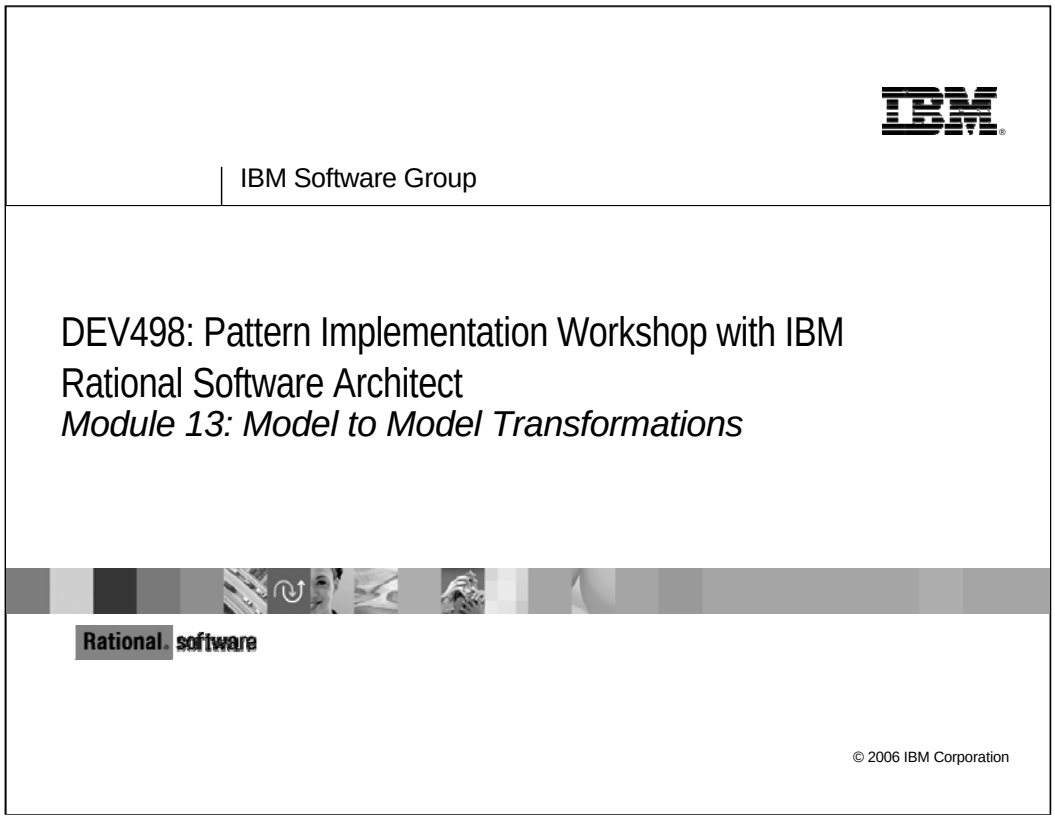
- Extending Rational Software Architect Functionality > Extending the UML metamodel by using custom UML profiles

**Web Resources**

- Simon Johnston. “UML 2.0 Profile for Software Services.” *IBM developerWorks*, http://www-128.ibm.com/developerworks/rational/library/05/419_soa/
- Kim Letkeman. “Comparing and merging UML models in IBM Rational Software Architect, Part 6: Parallel model development with custom profiles.” *IBM developerWorks*, http://www-128.ibm.com/developerworks//rational/library/05/0823_Letkeman/
- Duskco Misic. “Authoring UML profiles using Rational Software Architect and Rational Software Modeler.” *IBM developerWorks*, http://www-128.ibm.com/developerworks/rational/library/05/0906_dusko/index.html#N10452
- Bran Selic. “Unified Modeling Language version 2.0.” *IBM developerWorks*, http://www-128.ibm.com/developerworks/rational/library/05/321_uml/

**Literature**

- James Rumbaugh et al. *The Unified Modeling Language Reference Manual.* Boston: Addison Wesley, 2005.



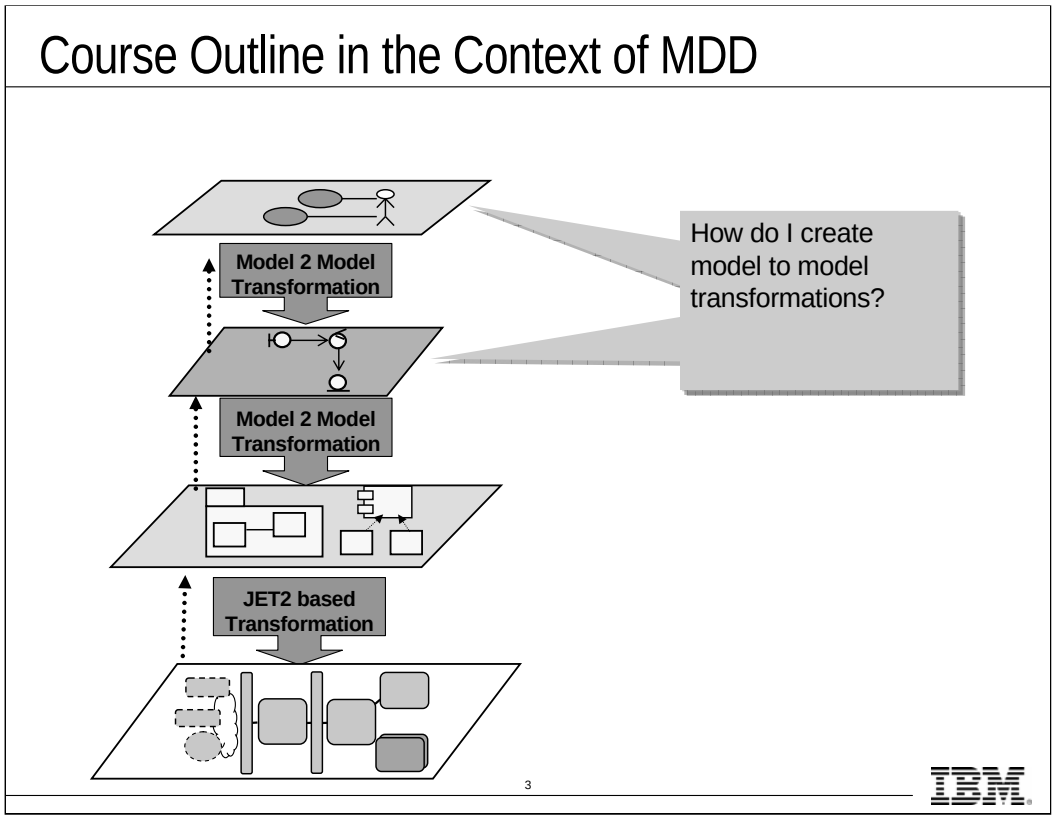

## Contents

## Model to Model Transformations

- Objectives:
  - Design a model to model transformation
  - Create a model to model transformation
  - Describe the ways in which model mapping and the resulting transformations can be combined

2

This module takes a closer look at model transformations in Rational Software Architect. Earlier, we saw how UML model transformations can be configured, and how the transformations can be used to connect a UML model to EMFT JET based transformation.




We will see this slide several times throughout the workshop. It will serve as a visual guide to the skills you are learning, and to how they fit into MDD .

## Where Are We?

- **Transformation Review**
- Model to Model using Transformation API
- Model to Model using Mapping
- Connecting Model to Model and Model to Text

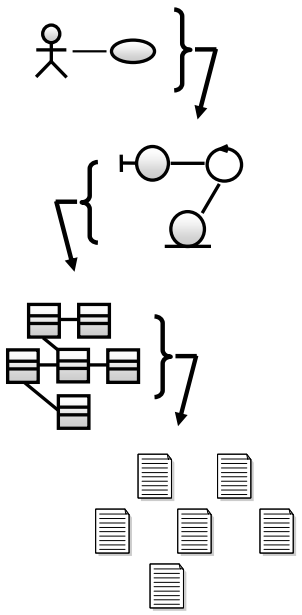

4

IBM

This section reviews some of the transformation concepts we've already covered.

## Review: Transformations

- Transformations create elements in a target model (domain) based on elements from a source model
- Often, the source domain is more abstract than the target domain
- Examples:
  - Based on a use-case model, create an analysis model containing analysis classes, sequence diagrams, and so on, that realize the use cases following company standards
  - Based on the analysis model, create a design model, containing the appropriate design classes, that incorporates elements of the company's security and persistence frameworks, and that follows the company standards
  - Starting with a UML model, apply Rational Software Architect's standard "UML to EJB" transformation to create EJB code elements



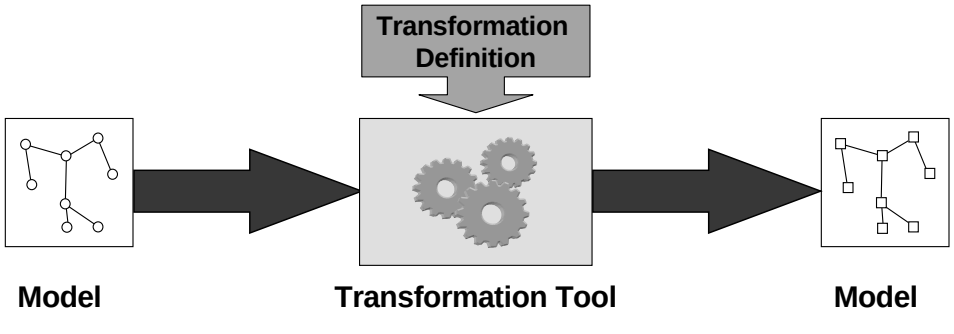
Transformations

5

IBM

## Review: Model Transformation Uses

- There is flexibility in choosing transformation sources and targets:
  - Transform a model to a model of the same type.
  - Transform a model across levels of abstraction.
  - Transform one type of model to another.
  - Extend another transformation.




6

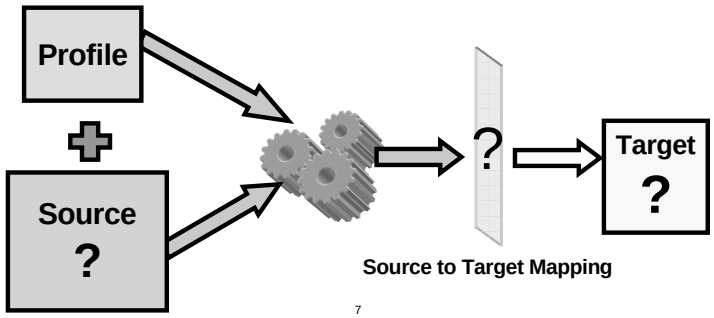
IBM

The following transformations are possible:

- **Across models of the same type:** When adding levels of refinement, you may transform from a PSM to another PSM. More details are added, but the type of model remains the same.
- **Across levels of abstraction:** Move from a PIM model to a PSM model as you add in details about the platform and get closer to the implementation.
- **From one type of model to another:** With transformations you can transform UML to code. This is the most common transformation available in Rational Software Architect.
- **Extend another transformation:** In Rational Software Architect transformations can be built on top of existing transformations.

## Planning the Transformation

- What is the source of the transformation?
  - Examples: use case model, analysis model, design model, and so on
- What is the target of the transformation?
  - Examples: analysis model, design model, and so on
- How does the source map to the target?
- How will the transformation work?
  - Size and complexity of input
  - Complexity of Transform process
  - Size and complexity of output




7

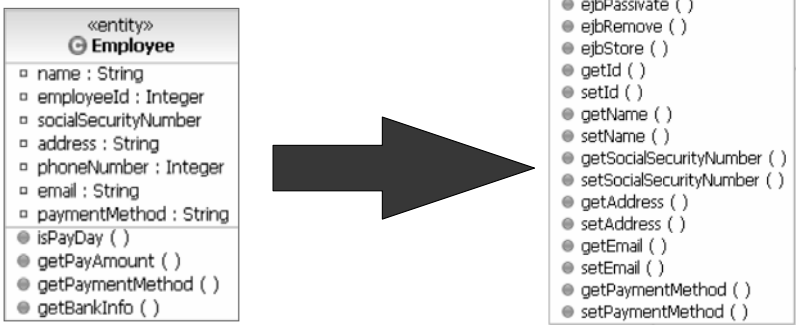
IBM

Before writing a transformation, determine the following:

- What will be the transformation source (information provider)?
- What will be the transformation target?
- What will the transformation generate?
- What information is required to generate the target?
- Where does this information come from?
- What is the format of the source information?
- What is the source-to-target mapping (are there any structural differences between the source and target models)?
- Which data in the source determines the created target?

# Planning: Metamodel and Semantics

- What are the semantics of an element?
  - What does it translate to?
  - Are there constraints?
  - Do I need additional data?
    - Flags, other annotations?
  - Which dependencies do they have?
  - Where is the information located in the source?
  - Where is the information located in the target?



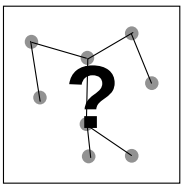

8

IBM

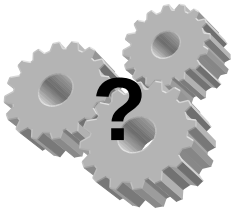
## Planning the Transformation: Determinations

Determine:

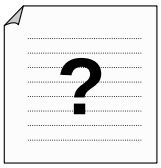
- Transformation rules
- Source navigation
- Transformation customization




**Source Navigation**




**Transformation Customization**




**Transformation Rules**

9

IBM

Other considerations in the planning of transformations include:

- What transformation rules are required?
- What is the source and target for each rule?
- How can the transformation be divided?
- What is the source navigation?
- How does the transformation provide each rule with its source?
- Can the transformation be customized?
- Can the transformation be configured?
- What transformation properties are needed?
- Can the transformation be extended?

# Steps in Creating a Transformation

- Plan the transformation
- Model and analyze the situation
- Drive decisions based on the results
- Model parts of the transformation
- Design parts of the transformation

10

IBM

## Where Are We?

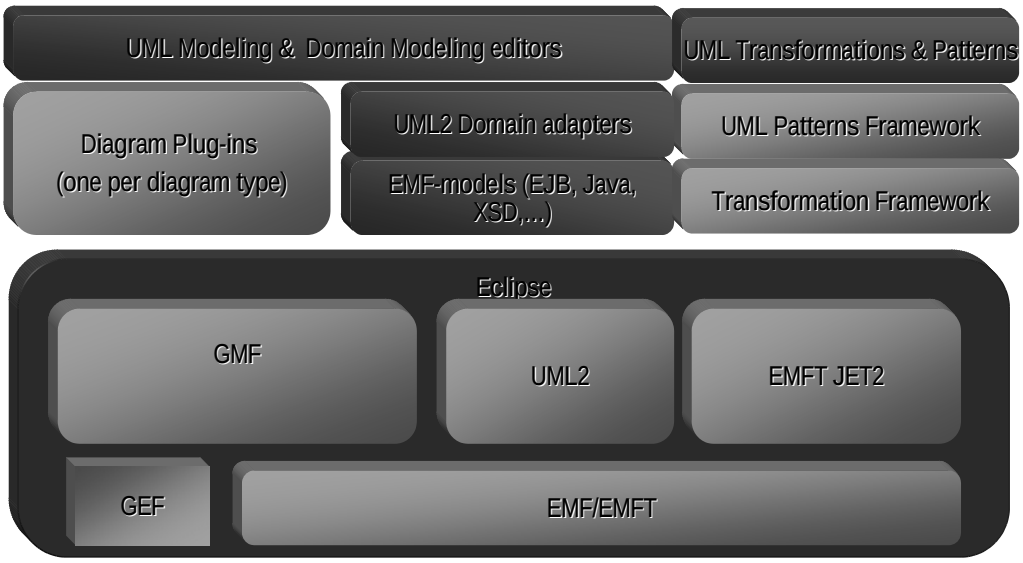
- Transformation Review
- **Model to Model using Transformation API**
- Model to Model using Mapping
- Connecting Model to Model and Model to Text


11

IBM

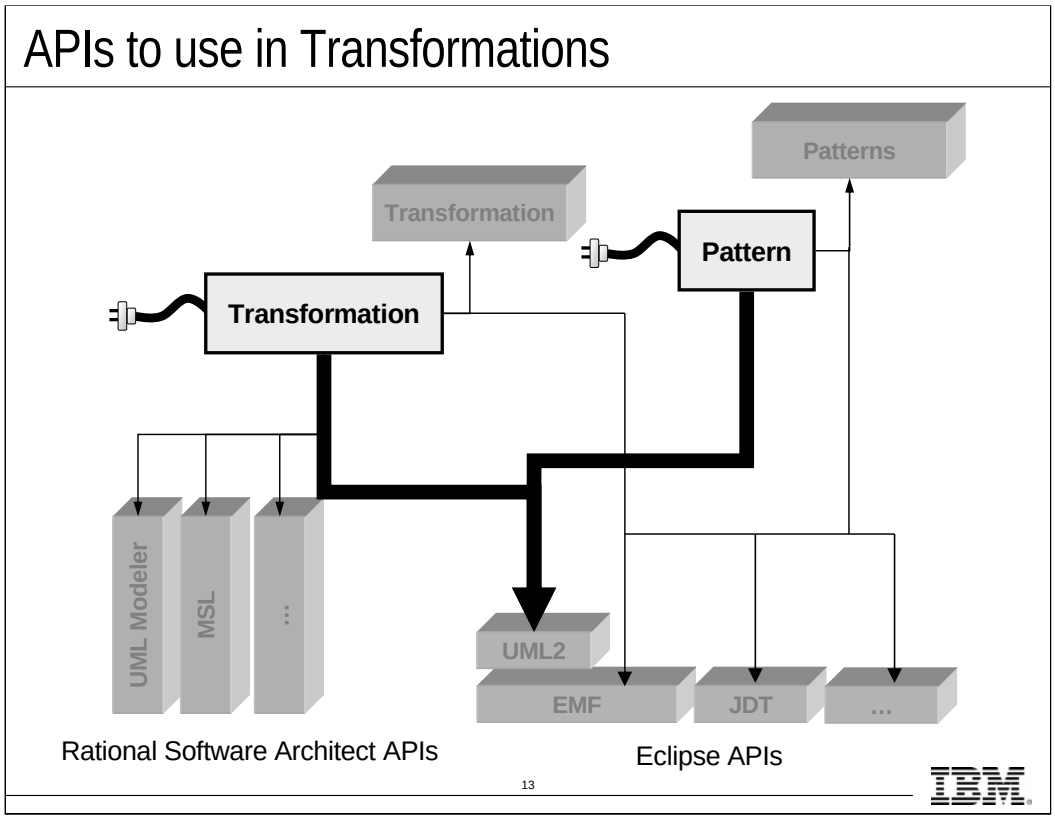
This section describes the steps to develop a transformation using the Transformation API.

# Rational Software Architect 7.0: Overview




12

IBM




When developing transformations, you should be aware of the following APIs:

- **Eclipse UML2:** The Eclipse UML2 API is an EMF-based implementation of the UML 2.0 metamodel. This provides us with an underlying structured data model for the models that we create within Rational Software Architect. Provides CRUD access to model elements. Supports all UML 2 user model objects and relationships (class, interface, package, association, dependency, generalization, and so on).
- **EMF (The Eclipse Modeling Framework):** Enables the Eclipse platform's modeling capabilities and code generation facility to interoperate with other tools and applications, for building tools and other applications based on a structured data model.
- **JDT** (**Java Development Tools):** Java development tools, along with the Eclipse technology, create applications that run on real-time operating systems and embedded environments.

The following APIs are available in Rational Software Modeler and Rational Software Architect:

- **UML Modeler:** A single utility class forms an API that exposes model and profile lifecycle operations, and provides access to the modeling platform.
- **MSL (Modeling Services Layer):** The MSL exposes classes and interfaces to manage Eclipse Modeling Framework (EMF) models.

## Elements of a Transformation

- Transformations contain the following elements:
  - **Transforms:** Containers that traverse the transformation element hierarchy
    - Execute extractors, rules, and nested transforms
    - Each transform gets its own parse through the source
    - Responsible for passing elements to rules
  - **Rules:** Responsible for transforming individual elements
  - **Extractors:** Responsible for extracting the next set of items from a given item, and passing them back to the transform( for example, All classes in a package)

14

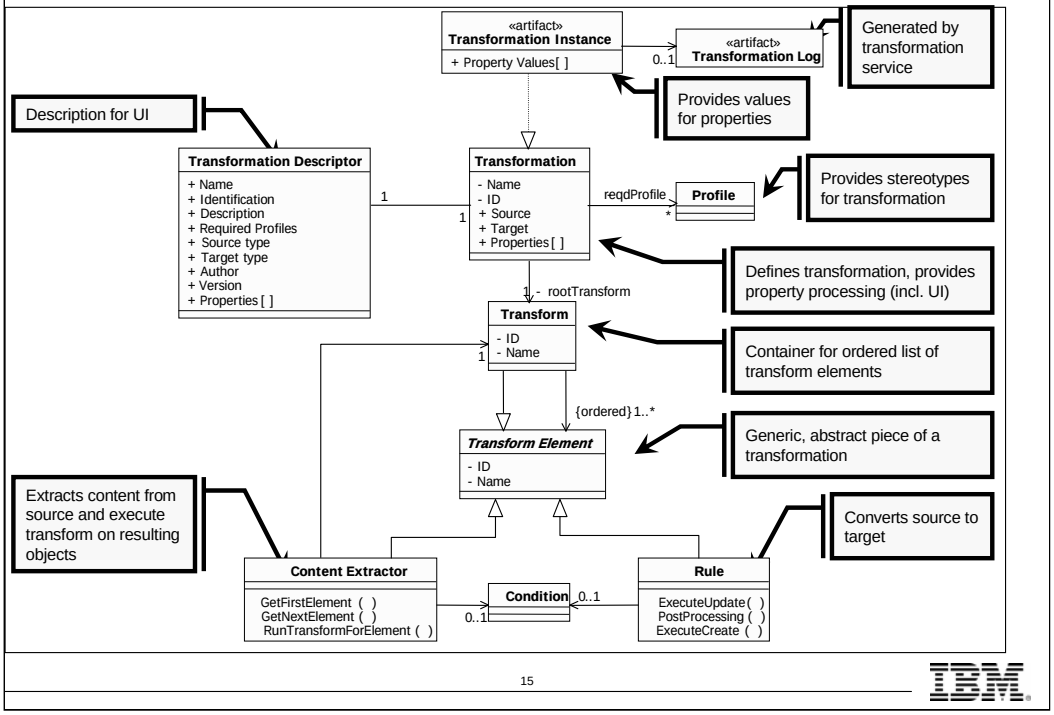
IBM

When you look at a model in Rational Software Modeler or Rational Software Architect, you see a visual representation of the model, with diagrams, packages, and classes. The visual elements simply represent a data structure of the model elements and their relationships. When thinking about an automated process like transformations, it is best to think of model elements in terms of data structures. The underlying structure of the model to be transformed is what matters.

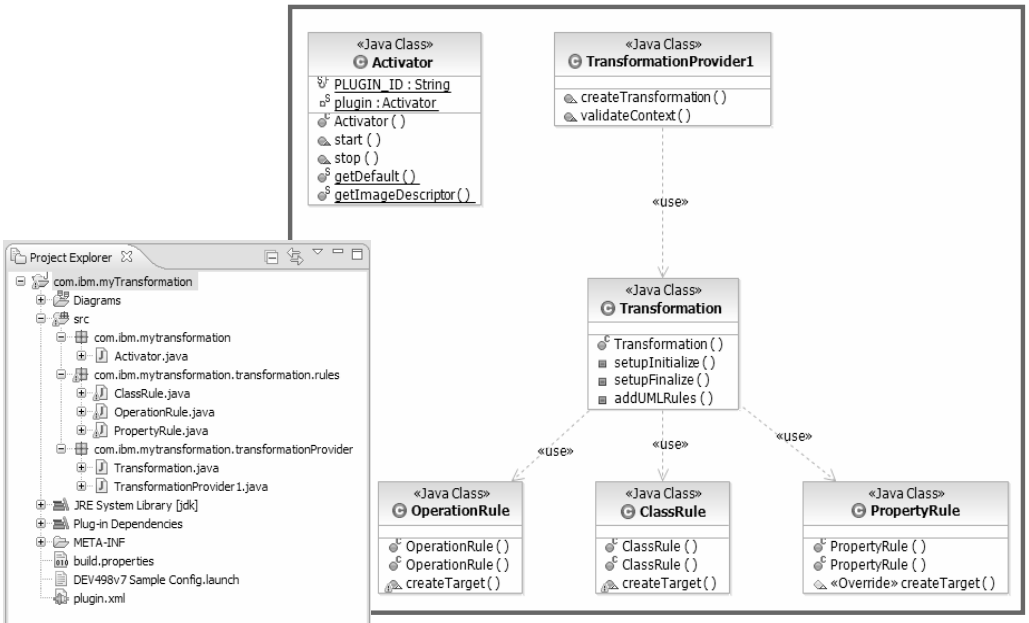
A similar case would be an XML document. You can open the document in Rational Software Architect (or any other XML editor) and view a user-friendly presentation of the underlying data. This makes it easy for you to review and understand both the data and the structure of the document. However, if you want to work with the data—either manipulating the data or using XSLT to transform the content—then it is the structure of the elements and the data that matter.

Transforms, rules, and extractors are types of classes in your transformation that will assist you in working with the source model, and in generating the target model. A transform contains a set of rules, extractors, and other transforms. When executing a transformation, the transform is called and will work with its extractors and rules to read data structure for the source model.

# Transformation Model



# Example Transformation Structure




The slide shows a simple example of a transformation created using the Plug-in With Transformation plug-in project template. In the diagram, you can see the classes that were described in the abstract on the previous slide: `TransformationProvider`, `Transformation`, and `Rule`. The other classes are part of the transformation engine or represent context elements outside the plug-in project.

## Steps to Write a Model to Model Transformation

To write a model to model transformation in Rational Software Architect:

1. Create a transformation plug-in project
2. Specify type of source and target model
3. Specify associated profiles
4. Add rules to the transformation
5. Implement transformation specific behavior
6. Test the transformation

17

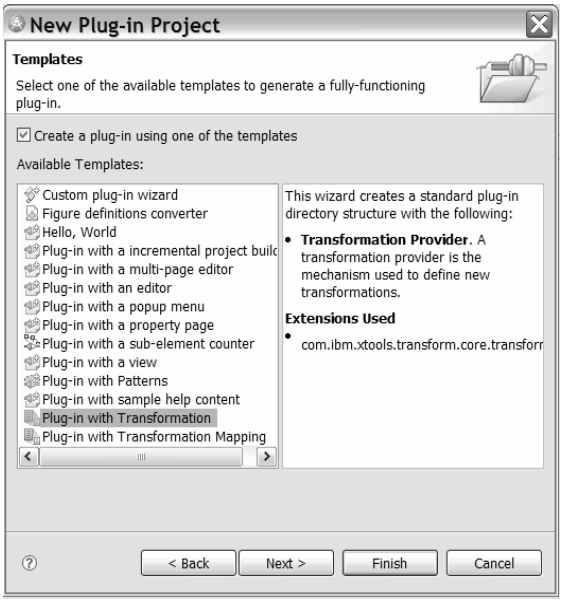
IBM

This slide shows the steps in creating a profile from scratch. Those steps are discussed on the following slides.

There will also be cases in which you will reuse the source for an existing transformation (for example some transformations ship in the sample gallery) or extend an existing transformation.

## Step 1: Create a Transformation Plug-in Project

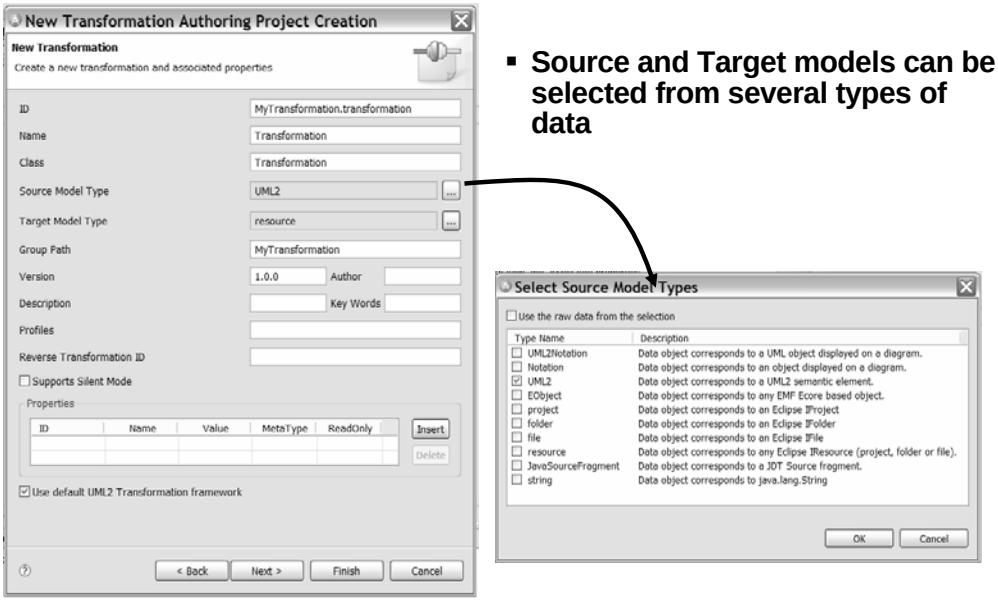
- Click **File > New > Generic Transformation Project**
- Enter a project name
- Select **Plug-in with Transformation** from the available list of templates




18

When creating a plug-in project, the Create Project Wizard includes a Plug-in with Transformation plug-in template that will produce the initial structure of a transformation to simplify the authoring effort. This structure includes directories for the plug-in class, rules, and transformation providers.

## Step 2: Specify Source and Target Model Type




The next page of the Wizard allows you to specify basic transformation properties.

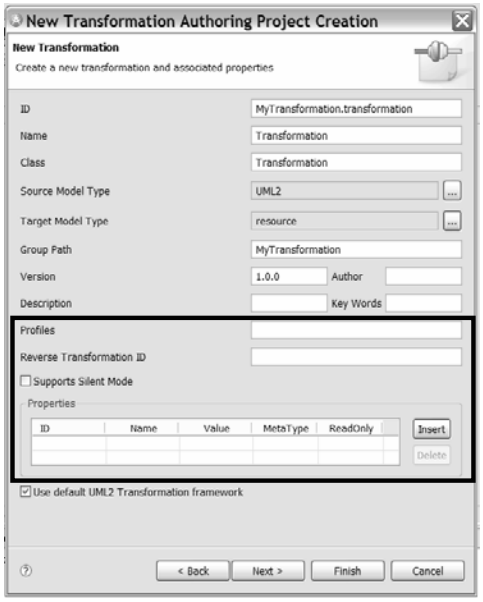
Source and target models allow you choose a category that will be used as a filter for the types of projects that can be used as either a Source Model or a Target Model. The category can be of one of the following types:

- **UML2:** Restricts the model to only those that contain UML2 elements.
- **UML2 Notation:** Restricts the model to only those that contain representations of UML2 elements based on the UML2 Notation API.
- **Resource:** Restricts the model to known Rational Software Architect project types.
- **Raw:** No filtering is applied.

# Step 3: Specify Additional Options

## Specify other options associated with the transformation

- Profiles from which rules of the transformation look for stereotypes
- Properties that supply additional configuration options to the transformation
- Support for silent running and reverse transformation



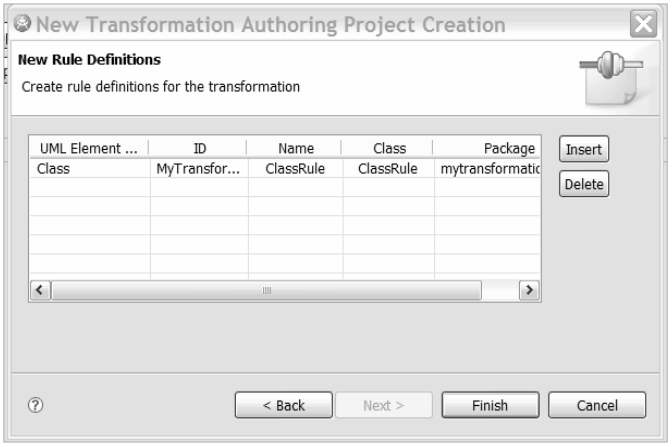

20

IBM

Specify any profiles associated with the transformation in the Profiles section of the New Transformation page. Rules that you create in the transformation will look for stereotypes from the profiles listed here when the transformation runs and make the appropriate changes in the target model.

# Step 4: Add Rules to the Transformation

- Create rules for any UML element type
- Name each rule
  - A class is created for each rule based on the rule name

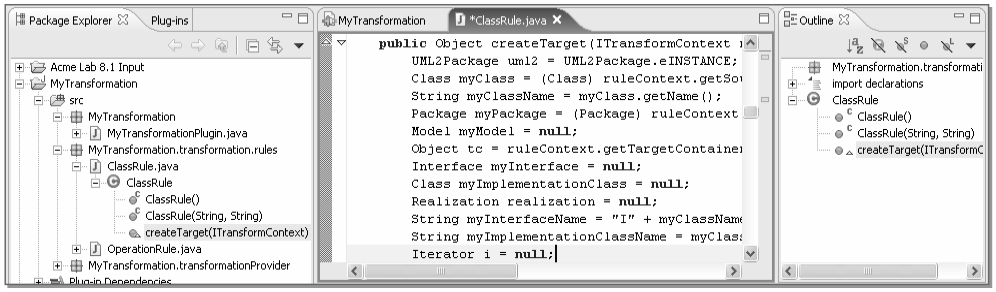



21

Conversion rules can be created based on any UML model element. Rules in a transformation convert one type of source element into one or more target elements. In addition to conversion rules, a transformation contains a mechanism to traverse the elements of the source model and to selectively run the appropriate rules based on the element type and rule-specific criteria. For example, a given rule might only run if the type of model element is a UML class that has a specific stereotype applied to it.

## Step 5: Implement Transformation Specific Behavior

- Implement Transformation Specific Behavior using **hotspots**
  - Hotspots are places in the transformation code where the transformation author can customize the transformation's behavior
- Numerous hotspots are available in the API




22

Working in the PDE, add code to your rules using the APIs made available in Rational Software Architect, which we have already discussed. The framework creates and positions methods in the code called hotspots. Hotspots are the significant non-final, public, or protected methods that can be overridden to alter framework or transformation behavior in some way at run-time. The `createTarget()` hotspot is discussed further on the following slide.

# The createTarget Method

- createTarget:
  - Is a key method in transformation rules
  - Creates a new target object based on source

```
public class ClassRule extends AbstractRule {
        public ClassRule() {
                super();
        }
        public ClassRule(String id, String name) {
                super(id, name);
        }
}
```

```
    public Object createTarget(ITransformContext context) {
        NamedElement element = (NamedElement) context.getSource();
        System.out.println("Class: " + element.getName());
        return null;
    }
```

23

When coding the rules for the transformation, the `createTarget` hotspot is of central importance. This is where information from the source model is converted into information for the target model. The `createTarget` method creates a new element if it does not exist. In cases where the element already exists you would not have to return anything.

# Accessing Profile Data: Source Code

```
protected void printQAProfileData (Class aClass) {
    out.println("Class: " + aClass.getQualifiedName());

    Stereotype qaStatus = aClass.getAppliedStereotype("QA::QAStatus");
    if (qaStatus != null) {
        out.println("  QAStatus.Approver: " + aClass.getValue(qaStatus, "Approver"));
        out.println("  QAStatus.Comments: " + aClass.getValue(qaStatus, "Comments"));
        out.println("  QAStatus.Test: " + aClass.getValue(qaStatus, "TestRating"));
        EnumerationLiteral approvalStatus  =
            (EnumerationLiteral)aClass.getValue(qaStatus, "ApprovalStatus");
        out.println("  QAStatus.ApprovalStatus: " + approvalStatus.getName());
    }
    else {
        out.println("  QAStatus not applied");
    }

    Stereotype retest = aClass.getAppliedStereotype("QA::Retest");
    if (retest != null) {
        out.println("  Retest.Reason: " + aClass.getValue(retest, "Reason"));
    }
    else {
        out.println("  Retest not applied");
    }
}
```

## Accessing Profile Data: Get QAStatus Stereotype

```
protected void printQAProfileData (Class aClass) {
    out.println("Class: " + aClass.getQualifiedName());

    Stereotype qaStatus = aClass.getAppliedStereotype("QA::QAStatus");
    if (qaStatus != null) {
        out.println("  QAStatus.Approver: " + aClass.getValue(qaStatus, "Approver"));
```

Determine if our stereotype has been applied

- *Element.getAppliedStereotype("Profile::Stereotype")*
  - returns the stereotype specified if it is applied
  - Specified with *"<ProfileName>::<StereotypeName>"*
- Additional methods
  - *apply(), unapply(), getAppliedStereotypes(), isApplied(), isRequired()*

25

## Accessing Profile Data: Getting QAStatus Properties

```
if (qaStatus != null) {
    out.println("  QAStatus.Approver: " + aClass.getValue(qaStatus, "Approver"));
    out.println("  QAStatus.Comments: " + aClass.getValue(qaStatus, "Comments"));
    out.println("  QAStatus.Test: " + aClass.getValue(qaStatus, "TestRating"));
    EnumerationLiteral approvalStatus  =
        (EnumerationLiteral)aClass.getValue(qaStatus, "ApprovalStatus");
    out.println("  QAStatus.ApprovalStatus: " + approvalStatus.getName());
}
else {
    out.println("  QAStatus not applied");
}
```

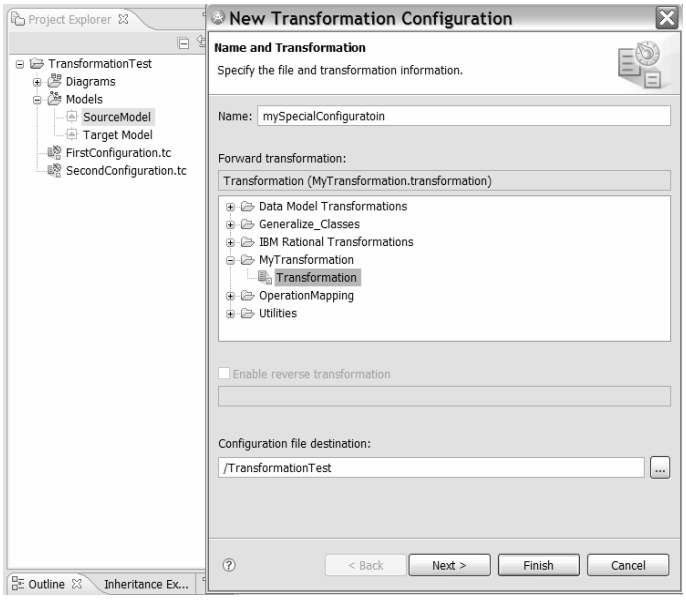
Property values can be explicitly accessed

- *<Element>.getValue(<Stereotype>, "<PropertyName>")*
  - Returns an object for the value of the property
  - Can print primitive types: *String, Integer, Boolean*
  - Enumerations need special treatment (*EnumerationLiteral*)
- *EnumerationLiteral* is a *NamedElement*
  - *getName()* returns name of literal

26

# Step 6: Test the Transformation

- Run the transformation in a runtime workbench
- Create source and target models to test the transformation
- Create a configuration of the transformation using the wizard
- The configuration is stored in a .tc file




27

From the PDE, launch a runtime instance of the workbench that will contain your transformation plug-in. From the runtime instance of the workbench, test your transformation by applying it to a model.

You can launch a runtime instance of the developer workbench to test the transformation. The developer and runtime instances are interactive. Any breakpoints or trace messages that you included in the transformation code are reported to the developer workbench.

**Note:** You cannot make changes to the transformation code while the runtime session is running.

To launch the runtime workbench:

1. Click **Window** > **Open Perspective** > **Other** to open the Select Perspective window.
2. Click **Debug** and click **OK**. The Debug perspective opens.
3. Click **Run** > **Debug.** The Debug window opens.
4. In the **Configurations** list, click **Run-time Workbench** and click **New**.
5. Type a configuration name in the **Name** field and select the **Clear workspace data before launching** check box to ensure that the latest changes to your pattern are used.
6. Click **Debug** to launch a new instance of the workbench.
   **Note:** After you set up a debug configuration, you can start a debug session by clicking the debug icon.

The Rational Software Delivery Platform splash screen appears while the run-time instance is loading.

## Demo: Create a Transformation

The instructor will now show you how to create a generic transformation

- Create a new plug-in project
- Add rules to the transformation
- Explore the packages and classes generated

28

IBM

Watch your instructor demonstrate how to create a simple transformation.

## Lab 10: Manually Create a Transformation (Optional)

- Given:
  - Code fragments
- Complete the following tasks:
  - Create a New Transformation Project
  - Add Rules to the Transformation
  - Create a Test Project
  - Run the Transformation
  - Add a New Rule

29

IBM

Complete Lab 10 in the student workbook.

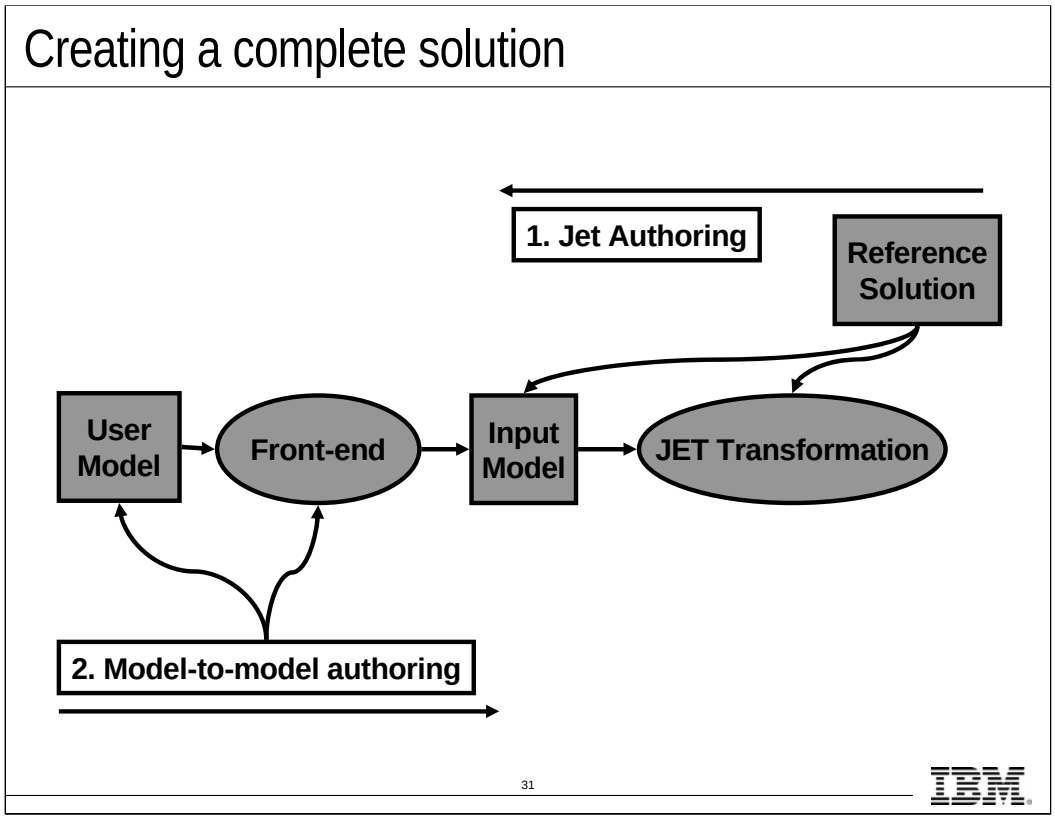
## Where Are We?

- Transformation Review
- Model to Model using Transformation API
- **Model to Model using Mapping**
- Connecting Model to Model
  and Model to Text


30

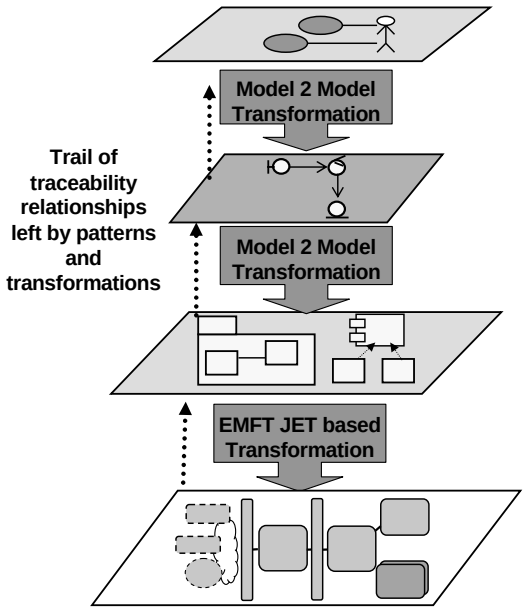
IBM

This section describes the steps to develop a Model to Model transformation using Mapping. Note that we will be using the same tooling as when we connected a UML model to an EMFT JET based transformation. As such, we'll take a more in-depth look at the Mapping options and features.




Using Rational Software Architect transformations, you can automate how you create and deliver software solutions. As shown on this slide, you are able to leverage Exemplar Authoring to quickly and easily automate how text based artifacts are created. In addition, you can leverage UML and Domain Specific Languages (DSL) using UML Profiles and EMF/GMF to create a front-end user representation. The Model mapping discussed earlier and again here can assist you in moving between levels of abstraction, either between models or from models to text.

# Ways to Drive Architecture: Classic MDD




Create the use-case model

Create the analysis model

Create the design model

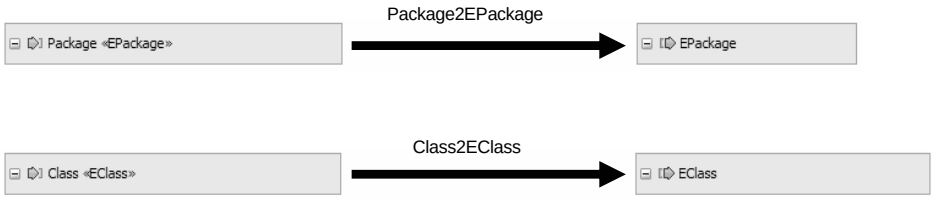
Complete the implementation using UML visualization

32

IBM

# Review: Mapping Models

## Mapping Models Contain Mapping Declarations




- Mapping models are Ecore models
- Mapping models contain references to the Ecore models that are being mapped, for example:
  - UML.ecore (input)
  - UML.ecore (output)
- Mapping models are persisted like other Ecore Models; they are serialized as XML files

33

# Review: Mapping Declarations

**Input Object**

| Package «EPackage» | |
|---|---|
| eAnnotations | EAnnotation [ ] |
| ownedComment | Comment [ ] |
| name | String |
| visibility | VisibilityKind |
| clientDependency | Dependency [ ] |
| nameExpression | StringExpression |
| elementImport | ElementImport [ ] |
| packageImport | PackageImport [ ] |
| ownedRule | Constraint [ ] |
| owningTemplateParameter | TemplateParameter |
| templateParameter | TemplateParameter |
| templateBinding | TemplateBinding [ ] |
| ownedTemplateSignature | TemplateSignature |
| packageMerge | PackageMerge [ ] |
| packagedElement | PackageableElement [ ] |
| profileApplication | ProfileApplication [ ] |
| packageName | EString |
| nsPrefix | EString |
| nsURI | EString |
| basePackage | EString |
| prefix | EString |

Package2EPackage

**Output Object**

| EPackage | |
|---|---|
| eAnnotations | EAnnotation [ ] |
| name | EString |
| nsURI | EString |
| nsPrefix | EString |
| eClassifiers | EClassifier [ ] |
| eSubpackages | EPackage [ ] |

- Mapping Declarations specify how to create or update an output object given an input object
- Mapping Declarations are named, for example, *Package2EPackage*

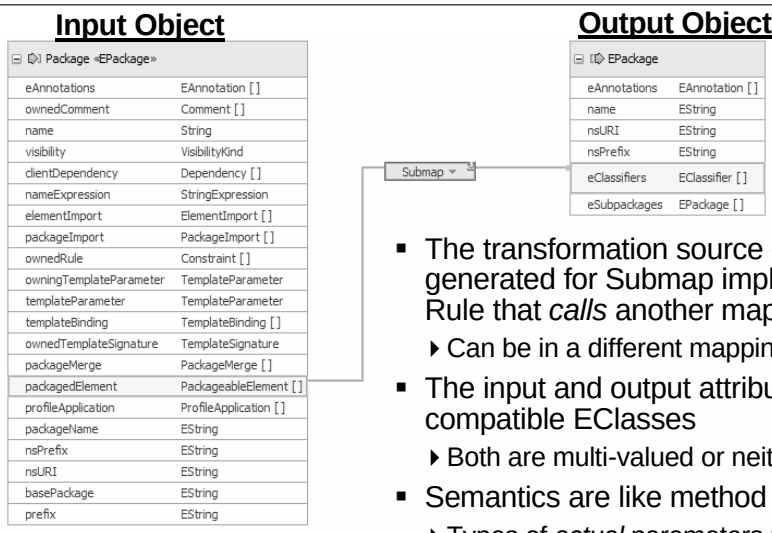
34

# Model to Model Mapping

- Let's take a more in-depth look at model mapping, including:
  - Custom mappings
  - Submap refinements
    - Condition
    - Input Filter
    - Output Filter
    - Extractors
  - Move refinements
    - Condition
  - Integration with Fuse

35

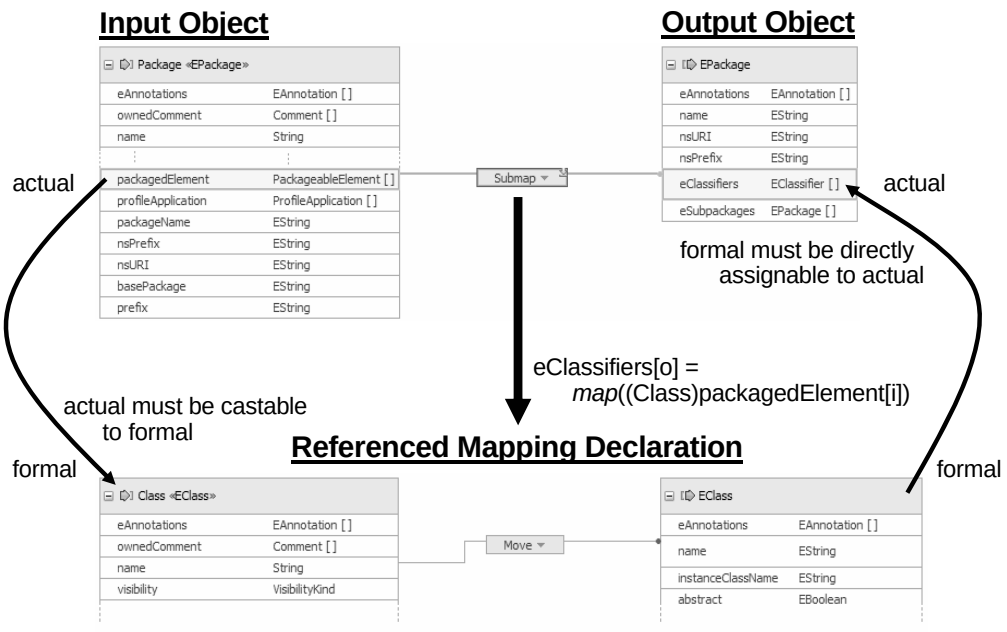
# Submap Mappings




- The transformation source code generated for Submap implements a Rule that *calls* another mapping
  - Can be in a different mapping model
- The input and output attributes must be compatible EClasses
  - Both are multi-valued or neither is
- Semantics are like method invocation
  - Types of *actual* parameters must be compatible with *formal* parameters
  - Can be recursive

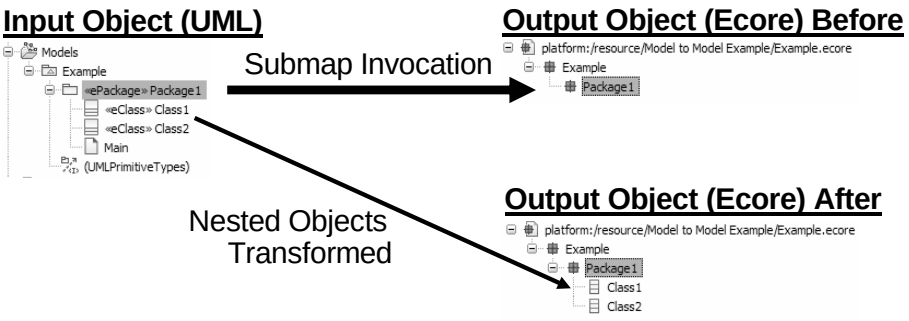
36

# Submap Mappings




37

# Submap Mapping Example

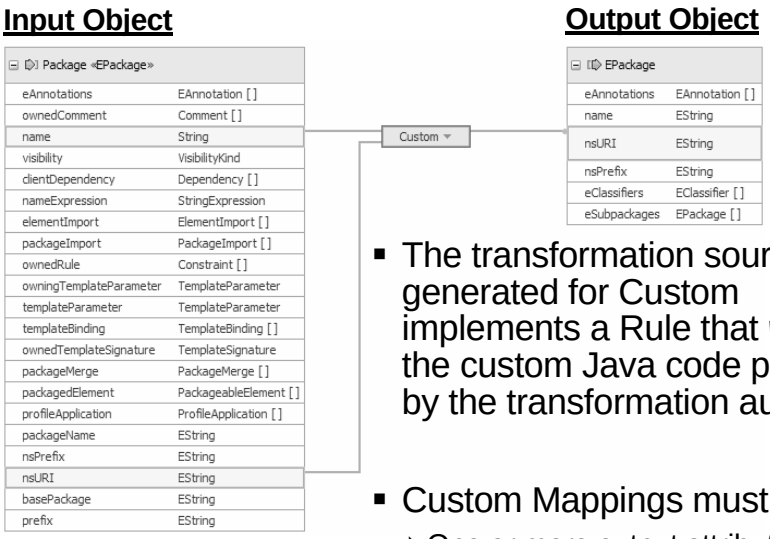



**Submap Invocation Pseudo-code**

```
elements = Package1_src.getPackagedElement();
for (i=0; i<elements.size; i++)
    If (elements[i] instanceof uml.Class) {              // filter input attribute
        uml.Class umlClass = (uml.Class)elements[i];     // adapt input attribute
        EClass ecoreClass = Class2EClass(umlClass);      // call referenced mapping
        Package1_tgt.getEClassifiers().add(ecoreClass); // add to output attribute
    }
```

38

# Custom Mappings




- The transformation source code generated for Custom implements a Rule that *wraps* the custom Java code provided by the transformation author
- Custom Mappings must have:
  - One or more output attributes
  - Zero or more input attributes
  - Zero or one input objects

39

# Custom Mappings: Java Source Code Snippets

- Java source code snippets for Custom mappings can be added directly into the custom mapping specification in the mapping file

```
// use specified URI if present and default to package name if not present
String uri = (String)UMLTransformAuthoringUtil.getStereotypeValue
                        (Package_src, "Ecore::EPackage::nsURI");
EPackage_tgt.setNsURI(uri!=null&&uri.length()>0?uri:Package_src.getName());
```

  - Code snippets are copied as-is into the generated transformation source code
  - Simple convention for naming variables
    - Input objects &attributes: <name>_src
    - Output objects &attributes: <name>_tgt
    - Duplicate names yield variable names like <name>_src2, etc.
  - Code assistance is provided in the Mapping Editor

# Custom Mappings – Java Classes

- Java classes that implement com.ibm.xtools.transform.authoring.RuleExtension can supply the Custom mapping processing

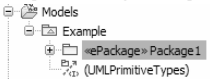
```
package uml_to_ecore_example.transforms;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EPackage;
import org.eclipse.uml2.uml.Package;
import com.ibm.xtools.transform.authoring.RuleExtension;
import com.ibm.xtools.transform.authoring.uml2.UMLTransformAuthoringUtil;
public class SetNsUri implements RuleExtension {
    public void execute(EObject source, EObject target) {
        Package umlPackage = (Package)source;
        EPackage ecorePackage = (EPackage)target;
        String uri = (String)UMLTransformAuthoringUtil.getStereotypeValue
                            (umlPackage, "Ecore::EPackage::nsURI"); //$NON-NLS-1$
        // use specified URI if present and default to package name if not present
        ecorePackage.setNsURI(uri!=null&&uri.length()>0?uri:umlPackage.getName());
    }
}
```

- Recommended over Java snippets in mapping file if:
  - Same (or very similar) processing can be used by multiple mappings
    (Avoids reuse via copy and paste)
  - You want the ability to change custom processing without changing mapping file

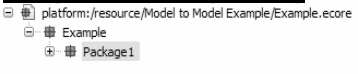
41

# Custom Mapping Example

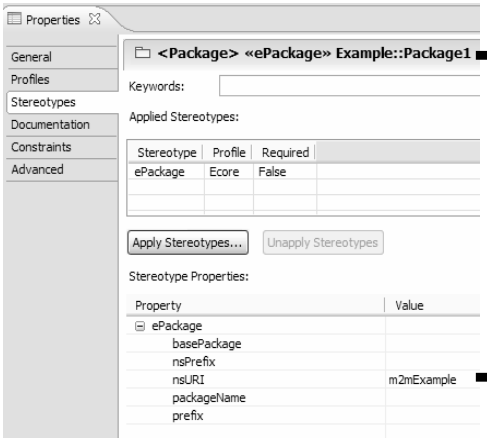
**Input Object (UML)**




**Output Object (Ecore)**

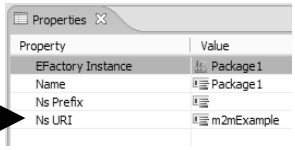



**Input Attribute**




Custom

**Output Attribute**




- The Custom mapping checks the input object's Stereotype attribute NsURI to see if it has been specified and, if available, assigns that value to the NsURI attribute of the output object. If it is not , the mapping available uses the name attribute value of the input object for the assignment.

42

IBM

# Mapping Refinements

- Refinements are optional customizations of mapping behavior
  - Implementation choices are similar to that used for Custom mappings
    - Java source code snippets
    - Java classes
- Refinement applicable to Move mappings
  - Condition
- Refinements applicable to Submap mappings
  - Condition
  - Input Filter
  - Output Filter
  - Custom Extractor

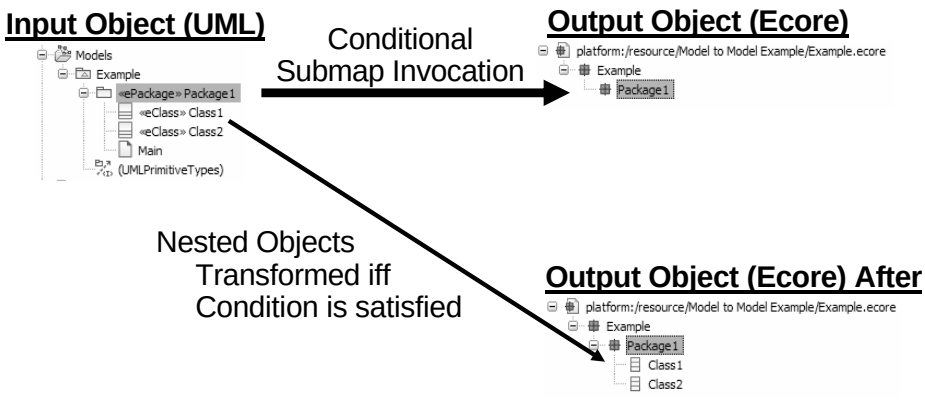
43

# Mapping Refinements: Condition

- Serves as guard on Move or Submap mapping execution
  - Execute mapping iff condition evaluates to true
- Java source code snippet implementation
  - Variable <name>_src designates the input object
  - Snippet must return a boolean value
- Java class implementation
  - Class must extend org.eclipse.emf.query.conditions.Condition
  - One method needs to be implemented
    - boolean isSatisfied(Object object)
      - The parameter object is set to the input object



# Condition Refinement Example




## Java code snippet for condition

```
// only process classes for public packages
return Package_src.getVisibility().equals
   (org.eclipse.uml2.uml.VisibilityKind.PUBLIC_LITERAL);
```

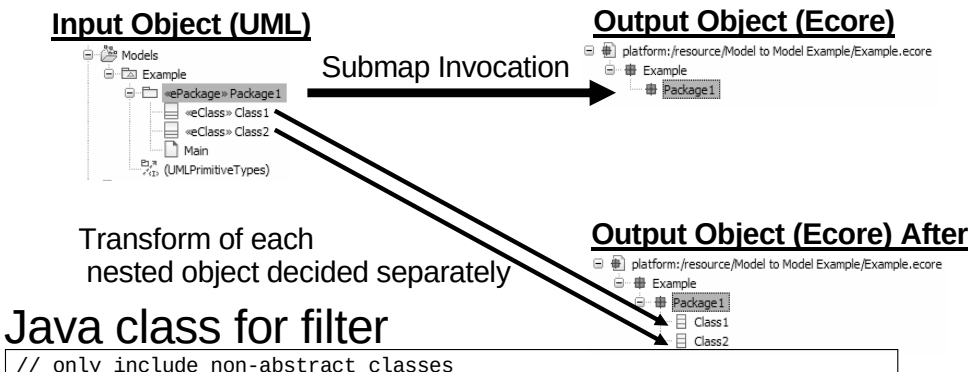
45

# Mapping Refinements: Input Filter

- For each value in the collection of the input object's attribute, designated in the submap, decide if that value should be transformed
  - Filter is called once for each value in the collection
  - Transform input value iff filter evaluates to true
- Java source code snippet implementation
  - Variable <name>_src designates the current value from the collection of the input object's attribute
  - Snippet must return a boolean value
- Java class implementation
  - Class must extend org.eclipse.emf.query.conditions.Condition
  - One method needs to be implemented
    - boolean isSatisfied(Object object)
      - The parameter object is set to the current value from the collection of the input object's attribute

46

# Input Filter Refinement Example




## Java class for filter

```
// only include non-abstract classes
package uml_to_ecore_example.transforms;
import org.eclipse.emf.query.conditions.Condition;
import org.eclipse.uml2.uml.Class;
public class IsNotAbstractClass extends Condition {
    public boolean isSatisfied(Object object) {
        return (object instanceof Class)?
            !((Class)object).isAbstract()
            :false;
    }
}
```

47

# Mapping Refinements: Output Filter

- Applicable only if the collection from the output object's attribute, that is designated by the submap, is not by containment
  - Non-containment implies that there could be duplicates in the collection
- For each value in the collection of the output object's attribute decide if that value should be transformed
  - Filter is called once for each value in the collection
  - Transform output value iff filter evaluates to true
- Java source code snippet implementation
  - Variable <name>_tgt designates the current value from the collection of the output object's attribute
  - Snippet must return a boolean value
- Java class implementation
  - Class must extend org.eclipse.emf.query.conditions.Condition
  - One method needs to be implemented
    - boolean isSatisfied(Object object)
      - The parameter object is set to the current value from the collection of the output object's attribute

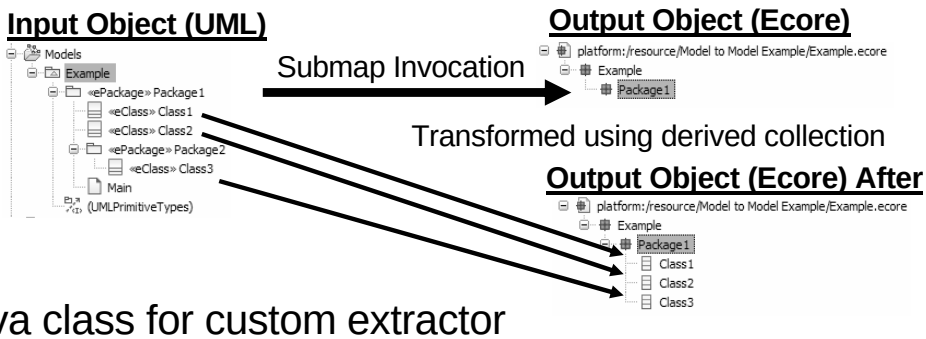
48

# Mapping Refinements: Custom Extractor

- Overrides the default extractor
  - Default extractor returns the collection from the input object's attribute that is designated by the submap
  - Override should return the collection of objects to be used when invoking mapping declaration designated in submap
- Java source code snippet implementation
  - Variable <name>_src designates the current input object
  - Snippet must return a java.util.Collection
- Java class implementation
  - Class must implement
    com.ibm.xtools.transform.authoring.ExtractorExtension
    - One method needs to be implemented
      - Collection execute(EObject source)
        - The parameter object is set to the current input object

49

# Custom Extractor Refinement Example




## Java class for custom extractor

```
package uml_to_ecore_example.transforms;
+import java.util.Collection;…
public class AddClassesFromNestedPackages implements ExtractorExtension {
    public Collection execute(EObject source) {
        Package pkg = (Package)source;
        Collection c = new BasicEList();
        // recursively add all classes in this package and its nested packages
        for (Iterator i=pkg.getPackagedElements().iterator(); i.hasNext();) {
            Object obj = i.next();
            if (obj instanceof Class) c.add(obj);
            else if (obj instanceof Package) c.addAll(execute((Package)obj));
        }
        return c;}}
```

50

## Typical Extension: Specify Model Merge Behavior

- The authored transformation outputs a model
  - The model need not be *complete*
- Transformation users will configure the transformation to place its output in a designated target container
  - The target container may or may not be empty when transformation runs
  - Configure **Rational Software Architect Model Fuse** to merge the new output with existing contents

```
<extension point="org.eclipse.core.runtime.contentTypes">
    <file-association
        content-type="com.ibm.xtools.comparemerge.emf.emfContentType"
        file-extensions="input"/>
</extension>
```

- Extension can be added to metamodel project or mapping project
  - Generic EMF merge is a good default
  - Specify the model type defined in the metamodel project

51

## Lab 11: Create a Model to Model Transformation

- Given:
  - Code fragments
- Complete the following tasks:
  - Create a New Transformation with Mapping Project
  - Create Transformation Mappings
  - Generate the Transformation Code
  - Create a Test Project
  - Run the Transformation
  - Add a New Mapping

52

IBM

Complete Lab 11 in the student workbook.

## Where Are We?

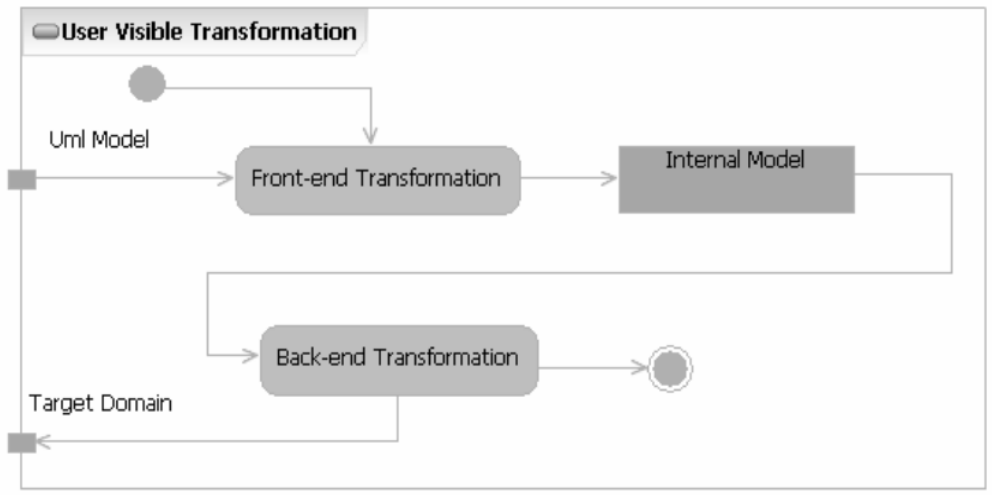
- Transformation Review
- Model to Model using Transformation API
- Model to Model using Mapping
- **Connecting Model to Model and Model to Text**


53

IBM

This section describes how the different models and transformations can be used in conjunction.

## Review: Activity Flow of a Typical Transformation

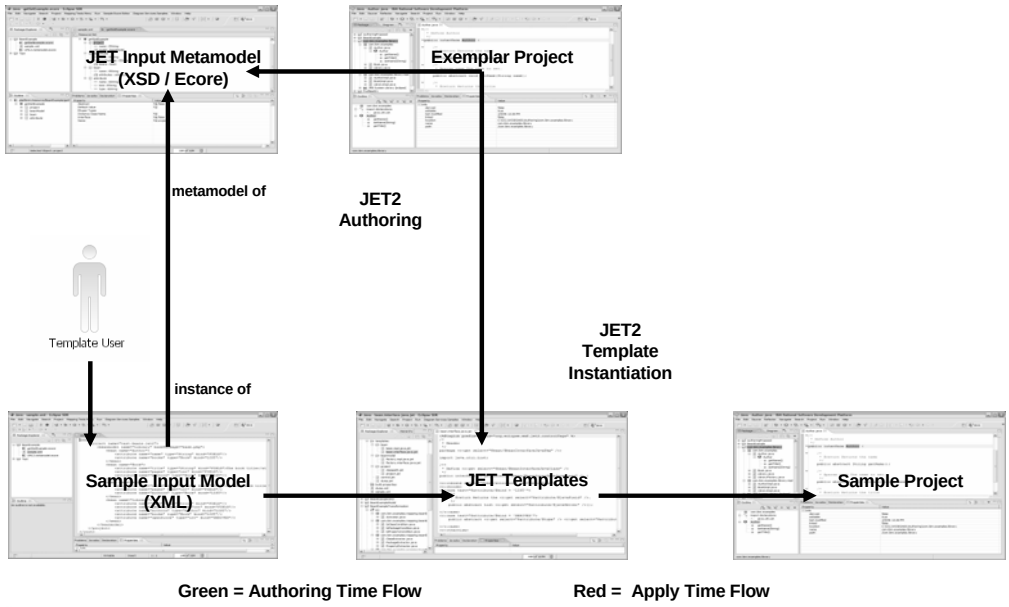



54

IBM

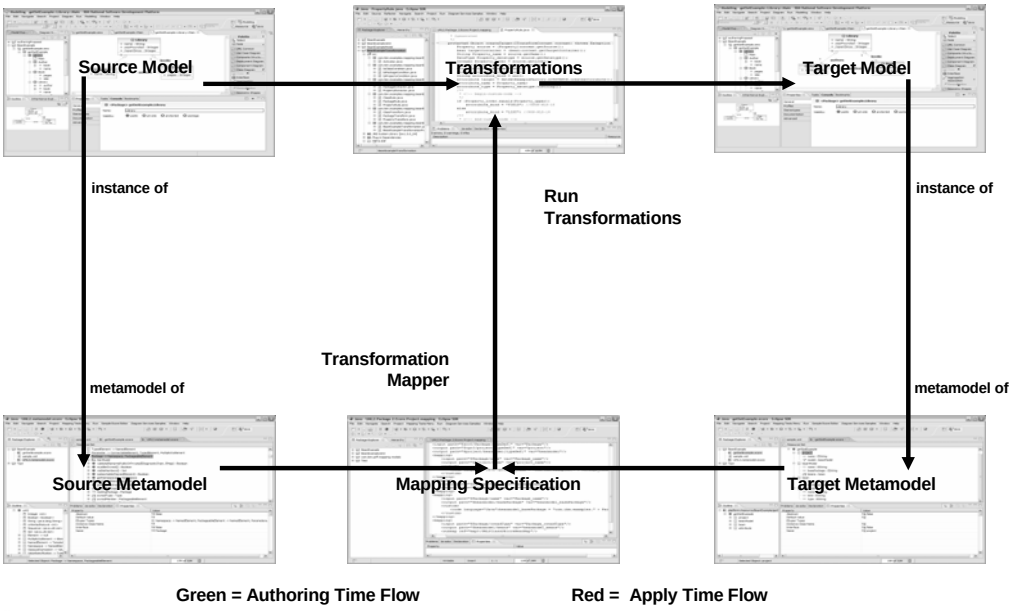
The top-level activity represents the transformation as seen by the user.

# Model to Text Transformations



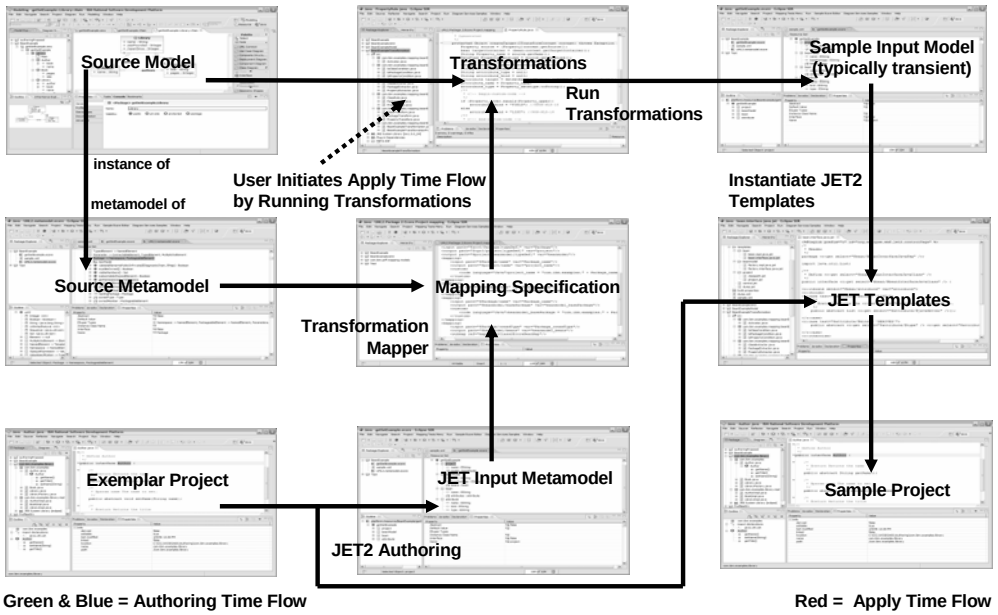

55

# Simplified Model to Model Transformations




56

## Model to Model and Model to Text Transformation Chain




57

# Review

- What are mapping refinements?
- What refinements are applicable to Submap mappings?
- What naming conventions are used when coding a custom mapping?



IBM

## Further Information

- Rational Software Architect Help
- Web Resources
- Literature


59

IBM

**Rational Software Architect Help Topics**

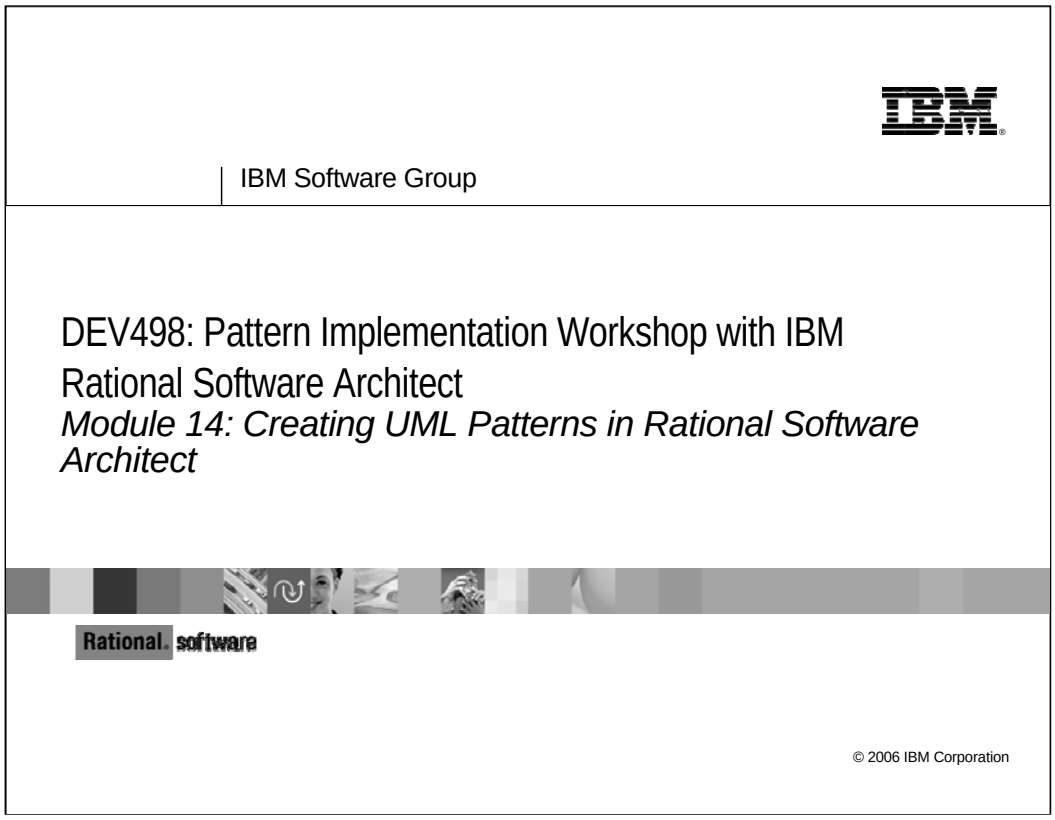
- IBM Rational Software Modeler API
- JET Tutorial Part 1 (Introduction to JET)
- JET Tutorial Part 2 (Write Code that Writes Code)
- Introduction to Transformation Authoring (from the Tutorials Gallery, an asset part of the developerWorks respository)

**Web Resources**

- API Documentation on the Eclipse UML2 component: http://download.eclipse.org/tools/uml2/javadoc/
- "Getting Started with UML2," http://dev.eclipse.org/viewcvs/indextools.cgi/%7Echeckout%7E/uml2-home/docs/articles/Getting_Started_with_UML2/article.html
- Alan Brown, "An introduction to Model Driven Architecture Part I: MDA and Today's Systems." http://www-128.ibm.com/developerworks/rational/library/3100.html
- Alan Brown, "An Introduction to Model-Driven Architecture Part III: How MDA affects the iterative development process" http://www-128.ibm.com/developerworks/rational/library/apr05/brown/

**Literature**

- Frankel, David S. *Model-Driven Architecture: Applying MDA to Enterprise Computing.* Indianapolis, IN: Wiley, 2003.



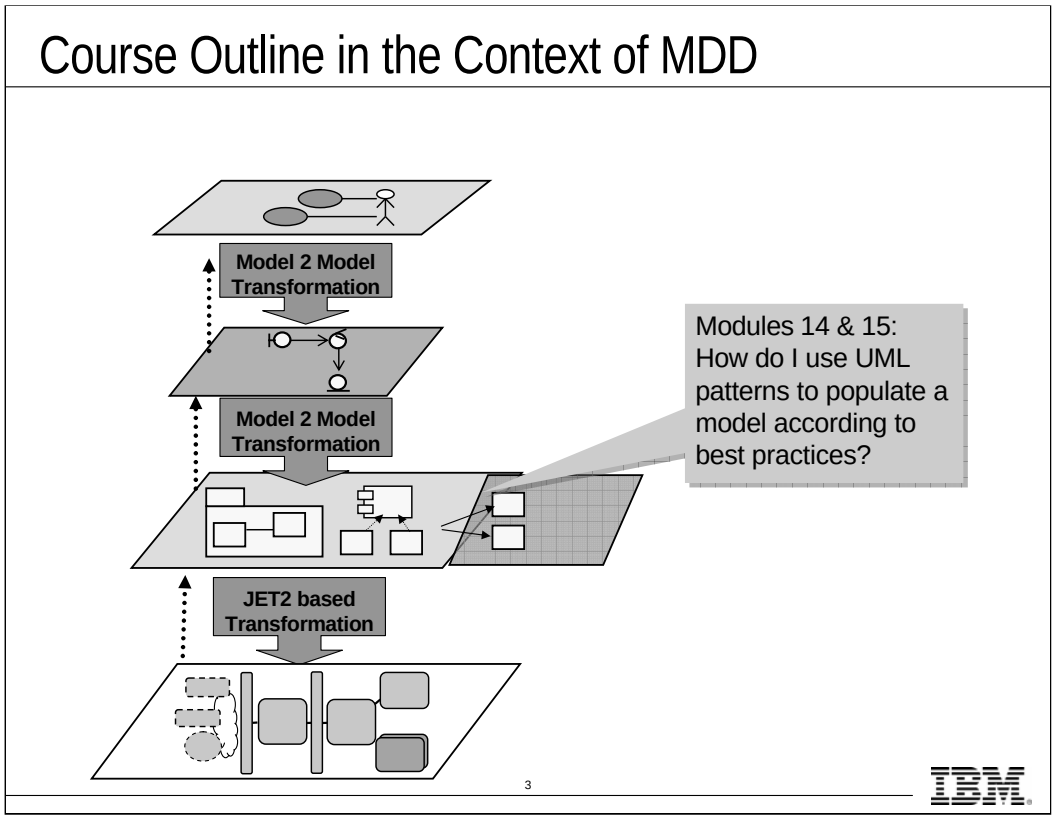

## Contents

## Creating UML Patterns with Rational Software Architect

- Objectives:
  - Describe the role of UML patterns in designing applications
  - Explain the process for authoring a UML pattern in Rational Software Architect
  - Create a simple UML pattern in Rational Software Architect

2

This module introduces UML pattern development in Rational Software Architect, including how to design and author UML patterns.




We will see this slide several times throughout the workshop. It will serve as a visual guide to the skills you are learning, and to how they fit into MDD .

## Where Are We?

- **Introduction to UML Patterns**
- UML Pattern Design
- Creating a UML Pattern in Rational Software Architect


4

IBM

This section defines and introduces UML patterns as a way to reuse and share software design solutions.

## Review: Patterns

- A pattern is a solution template for a recurring problem that has proven useful in a given context.
  - Can be used in all phases of development
- A pattern specification has:
  - A **problem** it solves
  - A **solution** for the problem
  - A **strategy** for applying the pattern in its context
  - **Consequences**, advantages, and disadvantages of implementing the solution.

5

IBM

Patterns provide a standard way of capturing and naming solutions, programming idioms, and best practices. As more developers research and understand patterns, patterns become a standard way for practitioners to communicate and share what they know with others.

For the designer, a set of carefully selected patterns, customized for a specific organization or project, can reduce time spent on repetitive tasks and help standardize approaches to specific design problems across projects and applications.

Pattern documentation is important. The pattern user does not need to know how to design a pattern, but good pattern documentation is needed for the pattern applier to locate, select, and apply a pattern. The user needs to know what problem the pattern solves, how it is solved, and the consequences of applying it.

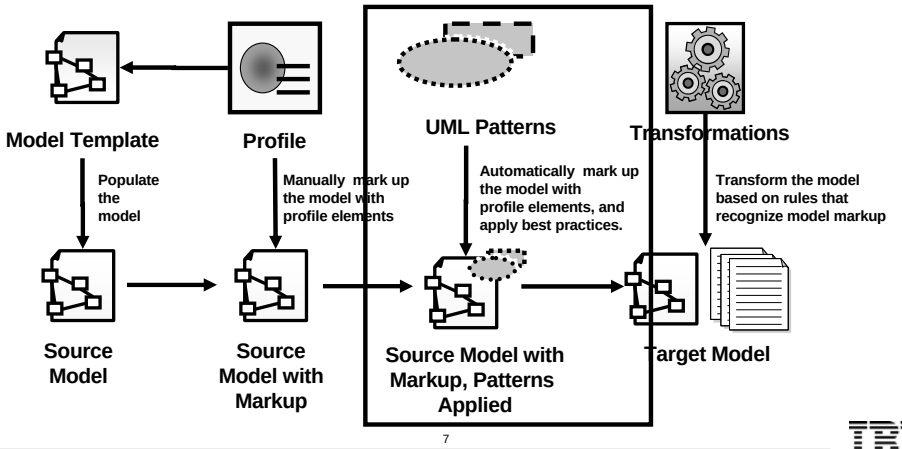
# UML Patterns vs. Transformations

- Model-to-model *in-situ* substitutions with a Rational Software Architect UML Pattern:
  - If you have a model where in-place model changes are desired
  - An example is applying the singleton design pattern to a class
  - Existing visual pattern authoring feature
- Model-to-model *rule-based* substitutions with a Rational Software Architect Transformation:
  - If you need to create a new model based on the content of an existing model
  - An example is the UML to Java transformation
- Model-to-text *exemplar-based* templates with a JET2 Template:
  - If you need to generate and manipulate textual artifacts based on model state

6

# The Role of Rational Software Architect UML Patterns

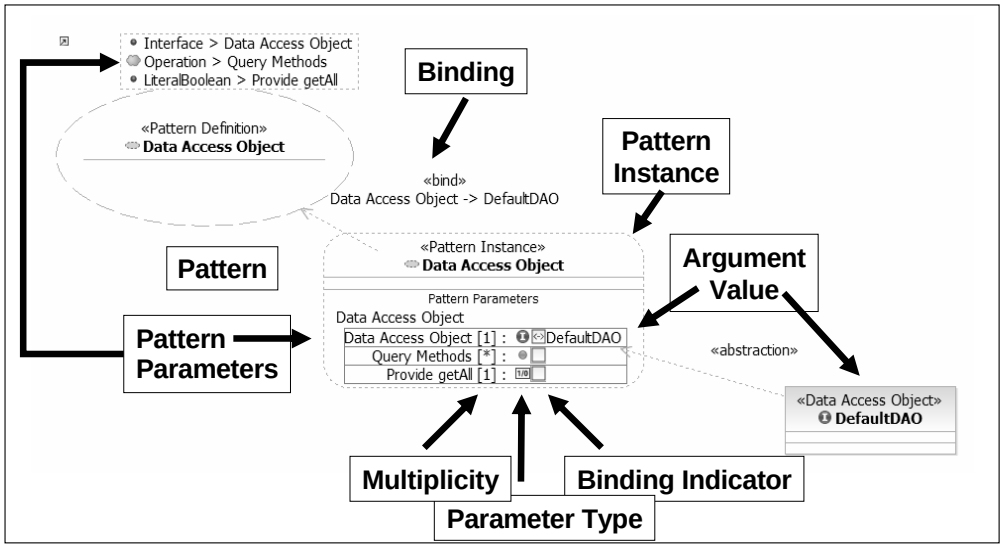
- In asset development, Rational Software Architect UML patterns help to develop the input model for a transformation.
  - Provide solutions at higher levels of abstraction
  - Provide standard ways to develop the solution
  - Can help ensure that profile elements are applied correctly and in a structure that makes sense




7

## Representing UML Patterns in Rational Software Architect

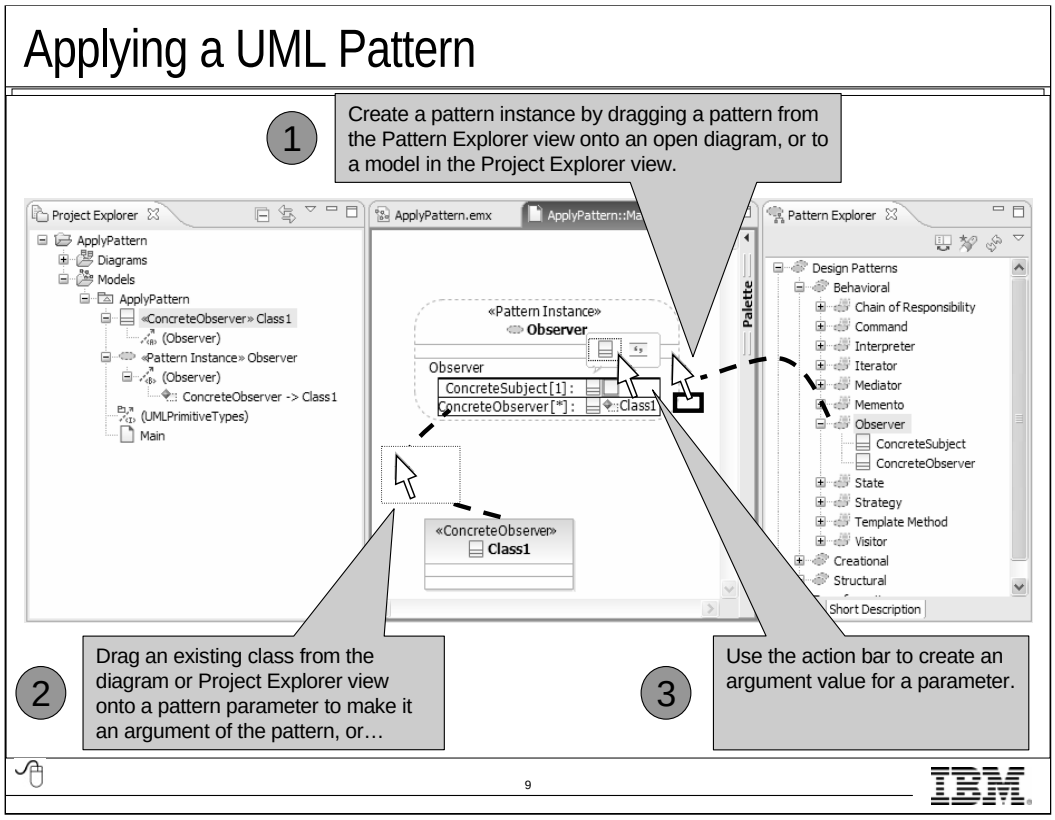
A UML Pattern is represented with a UML collaboration, package, or class




A UML pattern instance in Rational Software Modeler and Rational Software Architect is represented with a UML collaboration stereotyped «pattern instance».

The pattern instance includes the following features:

- **Parameters:** A UML pattern has points of variability, called parameters. Arguments are provided for parameters when the pattern is expanded into the target model. Each parameter in the pattern instance takes an argument. When the pattern instance is created, its parameters show the unbound parameter icon as an empty blue box. You can add or create an argument using the action bar, or by dragging an existing element from the diagram or Model Explorer view onto the parameter. When bound, the icon changes to a blue box containing a double arrow.
- **Parameter Multiplicity:** The parameter's multiplicity is shown in brackets after the parameter name.
- **Parameter Type:** After the multiplicity, an Eclipse-style icon or text shows the parameter type (for example, class, interface, or operation).
- **Binding Indicator:** An icon or text that shows whether the parameter has an argument bound to it. An empty blue box indicates that no arguments are bound to the parameter. A binding icon shows that arguments are bound to the parameter.
- **Arguments:** One (or more, if the pattern allows it) arguments can be bound to the parameter.




Applying a pattern is a two-step process: you first add an instance of a pattern to the model, and then select (or "bind") argument values for the pattern—either existing elements of the model or new elements that you create while applying the pattern.

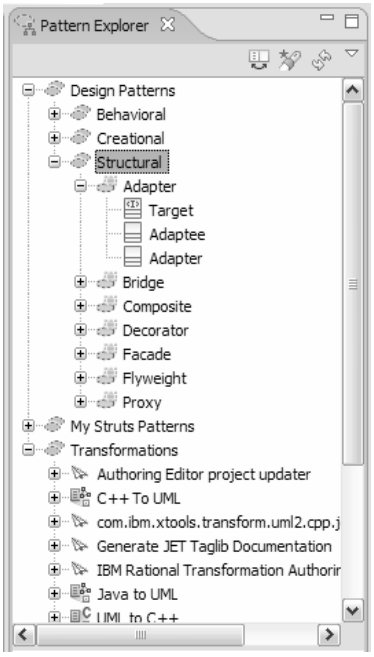
There are two ways to apply patterns in Rational Software Architect:

- Apply the pattern using the Apply Pattern wizard. Select a model as the location for the pattern instance and then select or create elements to use as argument values. You can continue to add argument values to the pattern instance after using the wizard.
- Apply the pattern interactively, using the Pattern Explorer view and the diagram editor. Drag the pattern from the Pattern Explorer and drop it onto an open diagram. If you click or hover the mouse over a parameter in the pattern instance, the action bar will appear, allowing you either to select an existing element in the model or to create a new one as the argument value. You can also drag and drop an existing model element, either from the diagram or from the Model Explorer view, onto a pattern instance's parameter to bind that element to the parameter.

To "unapply" a pattern, right-click the pattern instance and then click **Patterns > Unapply**. The pattern instance is deleted, and all bindings to model elements are deleted.

## Pattern Libraries

- Sets of related patterns are gathered into **pattern libraries**
- Patterns are sorted within libraries into **groups**




10

Patterns are always members of **pattern libraries**, and are always gathered into **groups** within the library. A pattern library is a collection of one or more patterns. Pattern libraries are implemented as Eclipse plug-ins, and each pattern is implemented by Java classes and XML files in that plug-in.

The groups that are shown in the Model Explorer view can be customized and rearranged. Right-clicking the elements in the Pattern Explorer provides you with options for creating new groups, renaming existing groups, and moving patterns between groups.

Rational Software Architect includes a number of patterns, including 23 GoF patterns to apply, categorized as Behavioral, Creational, and Structural patterns. In addition, there are eight GoF patterns to modify, including: Implementation, Interface, Keyword List, Directed Association, Delegation, Strategy, Singleton and Abstract Factory. These modifiable patterns must be imported into your installation from the Samples library: **Welcome page > Samples > Pattern Library**.

# Role of the Pattern User

- Recognize the modeling consistency and time-savings benefits when applying patterns to their UML models and therefore:
  - **Explore** the universe of available patterns to find a domain-specific pattern solution to the problem of interest
  - **Evaluate** the candidate pattern, and verify its applicability and usefulness to the problem of interest
  - **Apply** the selected pattern to the model, and incrementally select the pattern participants for each specified pattern role


11

IBM

## Where Are We?

- Introduction to UML Patterns
- **UML Pattern Design**
- Creating a UML Pattern in Rational Software Architect


12

IBM

This section introduces the process for designing UML patterns in Rational Software Modeler and Rational Software Architect, and discusses some issues to consider for effective UML pattern design.

## Identifying Opportunities for Pattern Creation

- Patterns are discovered, not invented
- Watch for recurring situations and solutions:
  - Explore the relationships between modeled classes in existing solutions
  - Review and inspect code
  - Review current literature
    - Books, articles, Web sites, and blogs that identify patterns particular to a specific interest area
  - Discuss problems and solutions with other architects, designers, and developers

13

IBM

Some suggestions for finding patterns:

- In existing models, use browse diagram to investigate the relationships between modeled classes.
- Review the current literature. Many books, articles, Web sites, and blogs identify patterns particular to a specific interest area.
- Identify repeating problems and solutions in code reviews and inspections.
- Follow discussions between architects, designers, and developers.
- Keep an eye out for repeating situations and solutions in your own work.

# Role of the Pattern Author

- Identify UML modeling pattern candidates and subsequently:
  - **Specify** the pattern in document form (also known as the pattern specification)
  - **Design and Implement** the pattern using Rational Software Architect visual authoring tools, which generate an Eclipse plug-in with Java code- and related OMG RAS pattern-profiled manifests
  - **Publish** the pattern, which involves *specifying* the pattern parts, *documenting* the pattern, *packaging* the pattern parts in a concise format, optionally *certifying* the pattern functionality and quality conformance levels, *distributing* the pattern, and *building awareness* around the newly offered pattern

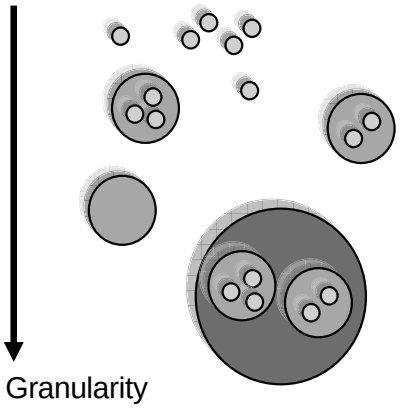
14

IBM

## UML Pattern Granularity

UML Patterns vary in granularity:

- **Micropatterns** are primitive type patterns.
- **Design patterns** are more abstract, and might reuse micropatterns or other design patterns.
- **Architectural patterns** are even more abstract, and might reuse design patterns or other architectural patterns.



Granularity

*Abstraction increases with pattern granularity*

15

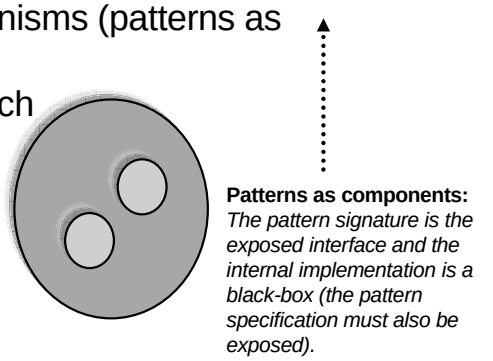
IBM

Patterns vary in granularity, from micropatterns to design patterns to architectural patterns.

- **Micropatterns** can be thought of as primitives because they cannot be decomposed into further patterns.
- **Design patterns** are larger in granularity and might be defined by reusing micropatterns and other design patterns.
- **Architectural patterns** are even larger in granularity and might be defined through reusing design patterns, other architectural patterns, or both.

## UML Patterns as Components In Pattern Authoring

- Consider pattern granularity and relationship to other patterns
- Because patterns in Rational Software Architect are implemented with Java code, apply common reuse strategies:
  - Patterns framework mechanisms (patterns as components)
  - Traditional OO methods such as composition or inheritance



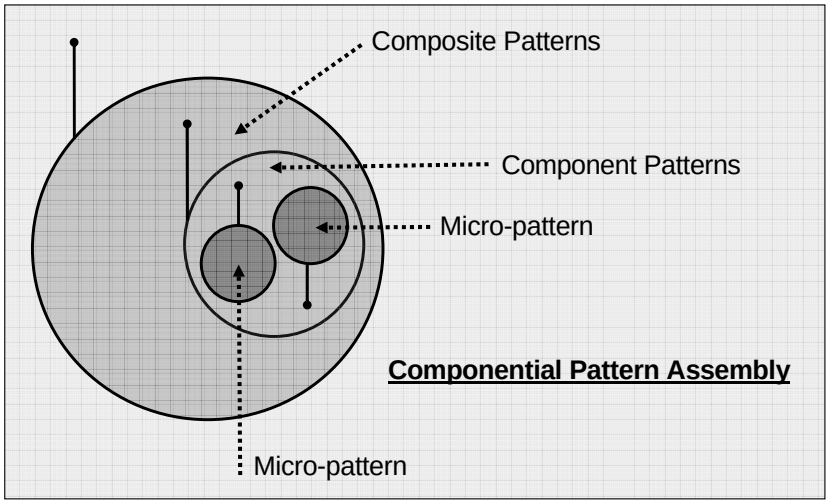

**Patterns as components:** *The pattern signature is the exposed interface and the internal implementation is a black-box (the pattern specification must also be exposed).*

16

When authoring patterns, reuse can benefit from thinking about a pattern's granularity and relationship to other patterns. Because patterns manifest as Java code, all development techniques used for application development could be used for pattern authoring as well. Patterns can be thought of as components from a pattern authoring point of view with the pattern signature being the exposed interface and the internal implementation being a black box (the pattern specification must also be exposed).

Reuse options available to the author include reuse using patterns framework mechanisms (patterns as components) or reuse using traditional OO methods such as composition or inheritance.

# Rational Software Architect Componential Patterns




17

## Where Are We?

- Introduction to UML Patterns
- UML Pattern Design
- **Creating a UML Pattern in Rational Software Architect**

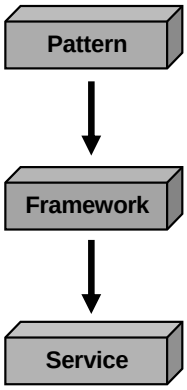

18

IBM

This section follows detailed steps for authoring patterns in Rational Software Architect.

## Pattern Service and Pattern Framework

- The Pattern Service acts as a broker for pattern clients
  - Discover definitions of patterns
  - Create instances of patterns
- The Patterns Framework:
  - Extends the pattern service to provide *default* pattern behavior
  - Can be used to add *expand* and *post-expand* behavior to the pattern




19

IBM

The Java-based pattern implementation model is created automatically by extending two plug-ins: a pattern service and a pattern framework that abstracts the use of the pattern service. Along with the Pattern Authoring view and the Pattern Explorer view in Rational Software Architect and Rational Software Modeler, the pattern service and pattern framework provide the basic functions to structure, design, code, search for, organize, and apply patterns.

The **Pattern Service** acts as a broker for patterns clients. It is responsible for helping the clients to discover patterns as well as create instances of patterns. In addition to being called when a user applies a pattern, the clients include patterns and transformations that expand nested patterns.

The **Pattern Framework** is a layer that operates between the pattern service and your pattern. The patterns framework provides the default pattern code for the pattern library, its member patterns, and their parameters. The framework promotes consistency across pattern libraries. The framework also provides for much of the processing that is common across patterns.

The pattern authoring tools generate Java source that makes calls into, and is called by, the framework. The pattern authoring tools generate Java source that makes calls into, and is called by, the framework. The main implementation task of a pattern author is to provide code for pattern behavior in the pattern's variability points, called hot spots.

## Key APIs and Classes

- EMF (Eclipse Modeling Framework)
  - Code generation facility
  - Based on a structured data model
- Eclipse UML2:
  - An EMF-based implementation of the UML 2.0 metamodel for the Eclipse platform
  - Provides CRUD access to model elements
  - Supports all UML2 user model objects and relationships (Class, Interface, Package, Association, Dependency, Generalization, and so on)
- AbstractPatternInstance:
  - Provides utility methods
  - Simplifies interactions with UML2 API

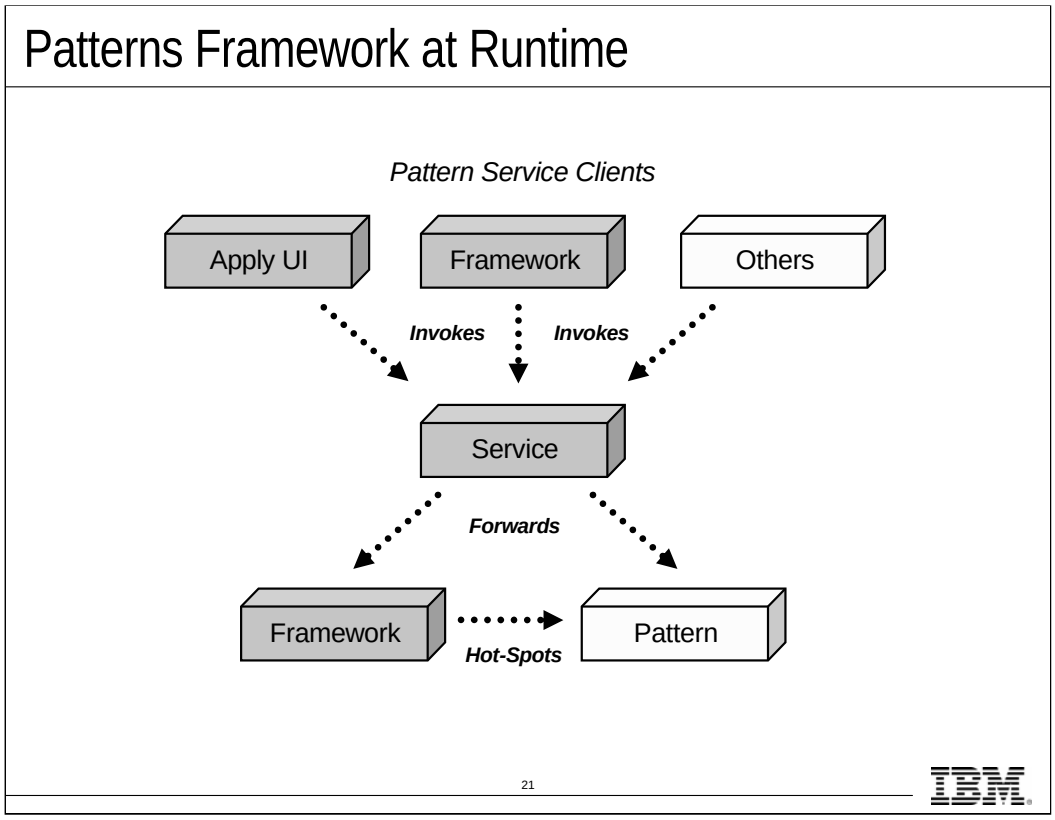
20

IBM

**EMF:** A modeling framework and code generation facility for building tools and other applications based on a structured data model. For the pattern author, it provides a structured object model that can be traversed and accessed.

**Eclipse UML2:** The Eclipse UML2 API is an EMF-based implementation of the UML 2.0 metamodel, providing the pattern author with an underlying structured data model for the models you create in Software Architect. You will work with UML2 elements whenever you need to examine what elements exist or modify the model as part of the pattern. For example, if you have a pattern that adds a method to a class you will need to work with the UML2 Class object, create a new Operation object, and then add it to the Class object.

**AbstractPatternInstance**: Provides utility methods for many common tasks when working with the UML2 API. In most cases, the utility methods provide an "intelligent" way to add information to an element in a model. In the example already cited, adding an operation to a class using the UML2 API, you would need to first check to see if the method exists. If the method does not exist, you would want to have the operation added, and if not, you would want to have no changes made. To make this check, you would make a single call to the `ensureOperation` method on `AbstractPatternInstance` and it will take the appropriate action.



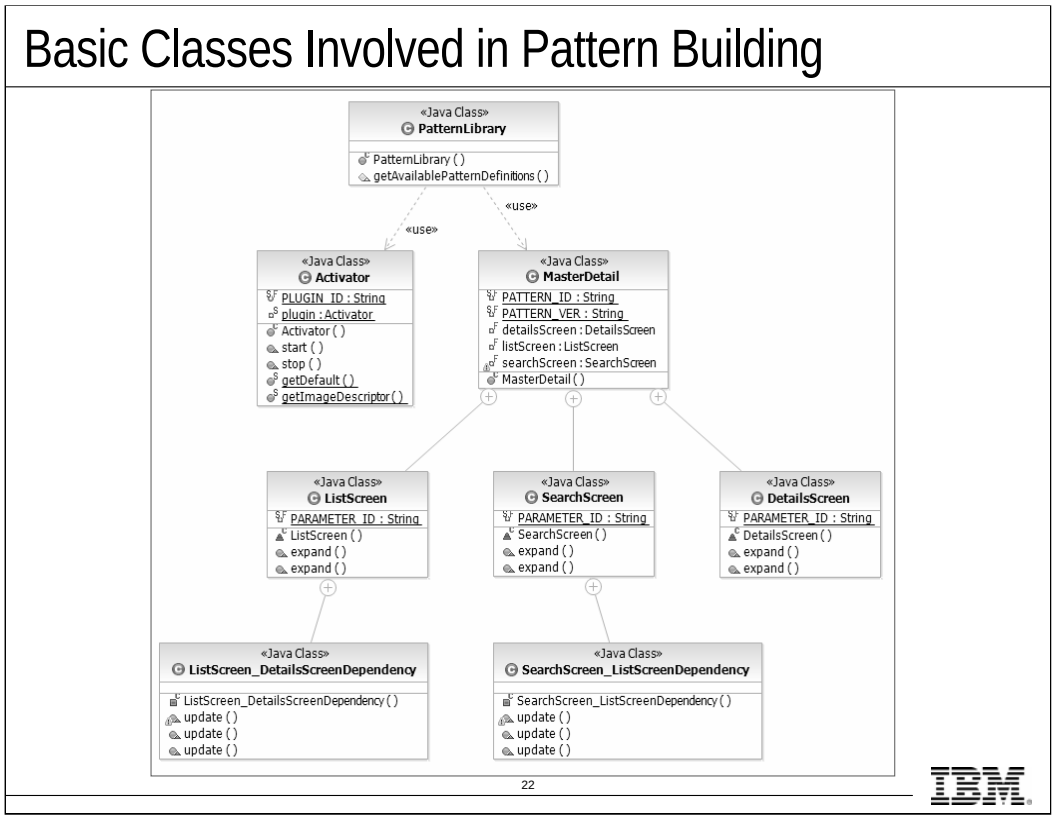

The pattern framework provides support for the base classes that are extended by the standard pattern implementation model that includes the pattern library, the contained patterns, and the pattern parameters. The framework promotes consistency in pattern design.

The framework is a layer between a pattern service and the end-user (generated and author written) pattern implementation. Pattern implementations depend on the pattern framework and the framework depends primarily on the pattern service. The service needs to know about what the pattern provides and provides interfaces and abstractions to define the contract of the pattern. The Framework must actually work with applying an instance of the pattern, so it needs to access the pattern behavior.

The pattern service discovers the available pattern plug-ins from a variety of sources, including installed plug-ins and local or remote repositories. The pattern service is also responsible for discovering pattern definitions, creating pattern instances, and directly supporting the client UI components. Both the pattern service and the pattern framework are Eclipse plug-ins.

Both UML 2.0 and RAS asset metamodels are supported within the pattern structure. A UML 2.0 representation of the pattern model is persisted in the pattern.

The default pattern model simplifies pattern authoring because the author must supply code only for the pattern executable behavior. The locations to add the expansion behavior are known as hot spots, and they are indicated by empty expansion methods. Dependent and independent expansion code is separated; hot spot update method locations are indicated to handle expansion dependencies when required by the pattern author.




Patterns are composed of the following basic classes:

- **Pattern Library Class:** The outermost abstraction that contains pattern definitions. The PatternLibrary class in a pattern is a subclass of the AbstractPatternLibrary class, which is a façade for nearly all invocations forwarded from the pattern service.
- **Pattern Definition Class:** Contained within a pattern library and instantiated at run-time by the pattern author's concrete library.
- **Parameter Class:** When the pattern author adds parameters to the pattern, they are added as inner classes (shown in the class diagram with the owned element association) to the pattern definition class and are instantiated when the pattern is applied by the pattern definition class's constructor.
- **Dependency Class:** Parameter classes have inner classes for observing dependents and observed dependencies, which is typically instantiated by the owning parameter's constructor.

## Pattern Delegate Mechanism

- The Pattern Delegate Mechanism allows you to apply one pattern from within another
- Key Classes involved are:
  - PatternDefinitionUsage
  - PatternDependencyDelegate

23

IBM

As discussed previously, there will be times when you find that there are other patterns that implement some behavior that you want to reuse within a new pattern that is being created. Rather than rewriting, or copying and pasting the code, you can delegate responsibility to the other pattern.

Key classes:

- **PatternDefinitionUsage:** Represents the use of a pattern definition. A pattern definition usage is required when constructing a pattern delegate instance. When you want to have another pattern used within your pattern, you will first need to create a PatternDefinitionUsage instance that refers to the pattern that you would like to utilize within your pattern.
- **PatternDependencyDelegate:** The delegate enables pattern dependency implementations to move some of the behavior that a dependency is responsible for into another pattern.

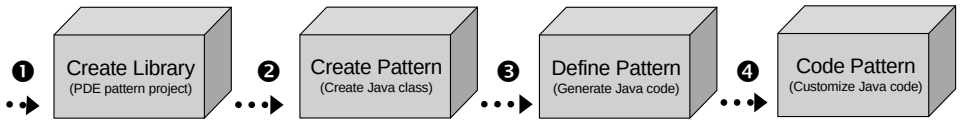
# Four Steps of Pattern Creation

To create a pattern:

1. Create a pattern library
2. Add a pattern to the pattern library
3. Define the pattern
   - Add template parameters to the pattern
   - Specify dependencies between template parameters
4. Implement pattern specific behavior

*Using Authoring UI*

*Using Java Editor*




24

IBM

Details on the steps in the above slide:

A new pattern project is created to contain created patterns and represent the pattern library, the appropriate libraries are adjusted in the `plugin.xml`, and a manifest is created.

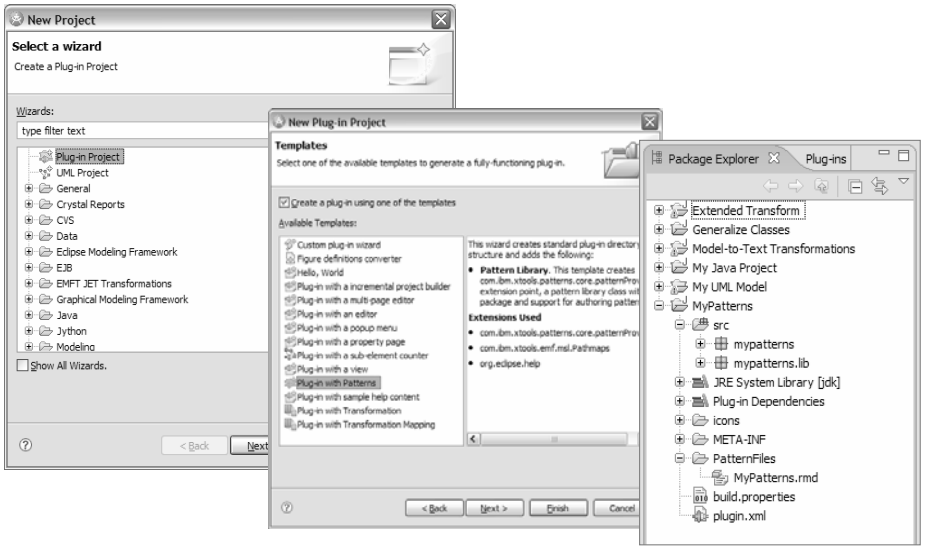
A new pattern is created using the pattern authoring UIs, Java classes are created to represent the pattern, and a manifest is created.

A pattern is structured with the UI by creating signature and parameter dependencies, a Java inner class is created for each parameter and an inner class for each dependency, and the manifests are updated.

Additional code is added to the generated Java classes, implementing patterns framework hot spots (the primary hot spots are expand and update), pattern delegates are coded, and manifests are adjusted if necessary.

## Step 1: Create a Pattern Library

- Create a plug-in Project
- Select the plug-in with Patterns template




25

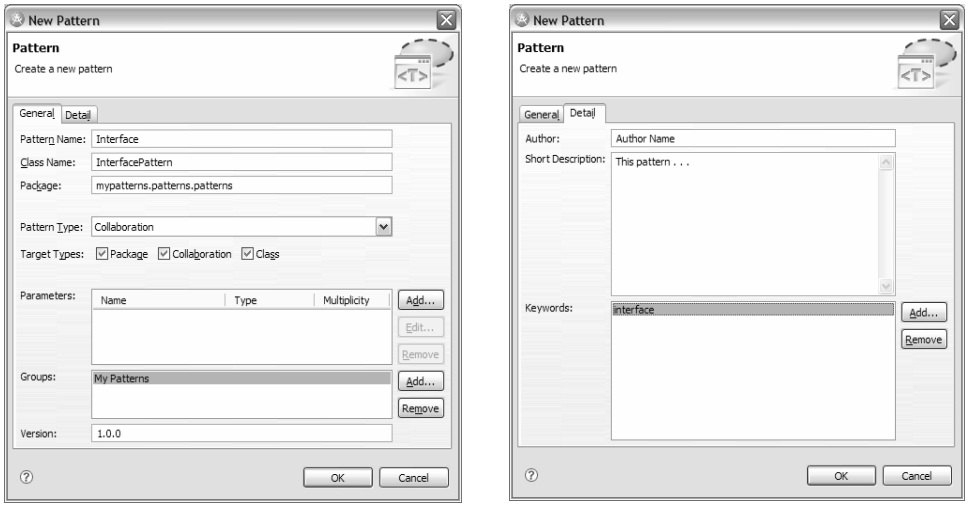
IBM

The example used in this section is the simple interface pattern used in the Rational Software Architect cheat sheet on pattern authoring. The Interface pattern has two template parameters: an interface with methods and a class that implements them. The pattern ensures that the class implements each of the methods on the interface. The apply-time behavior of the pattern includes adding an implementation relationship and a set of methods bodies to the class.

Begin by creating a new pattern library. Pattern libraries are implemented as Eclipse plug-ins. Rational Software Architect also has a plug-in template available to provide a quick start for pattern authoring.

# Step 2: Add a New Pattern to the Library

## Name and describe the pattern




26

IBM

A pattern definition is created when the pattern author uses the Pattern Authoring view to add a pattern and its template parameters to a pattern library. Each pattern in the pattern library has its own pattern definition.

The pattern definition is a UML 2.0 element with a keyword of Pattern Definition. Depending on the pattern type, it is a parameterized collaboration, class, or package.

The pattern author can locate the pattern definition in the pattern definition model (the plug-in project used to create the pattern). However, the author should not directly modify the pattern definition. A new pattern definition can be regenerated if it gets out of sync with the pattern's Java code.

In the pattern application process, the pattern definition provides essential model information. Although not readily visible to the pattern applier, the pattern definition is bound to all generated pattern instances.

**Pattern templates**

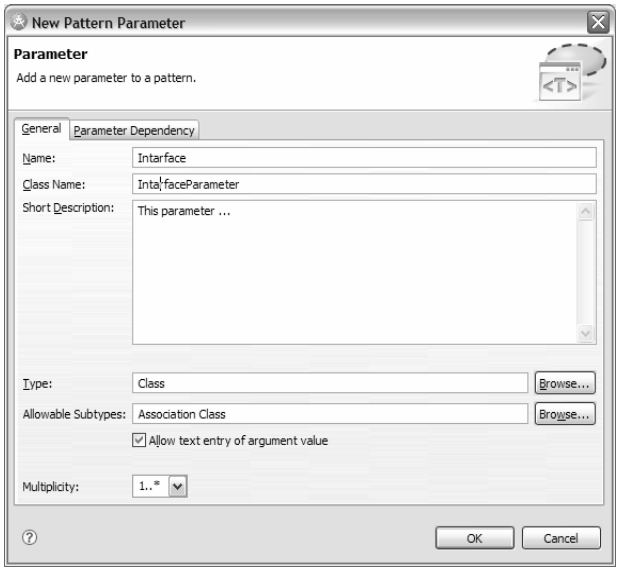
The pattern framework supplies default code for each pattern as you add each pattern to the library. You can modify this code, except where comments indicate that modifications will render the pattern incompatible.

**Pattern Template Parameters**

The pattern framework supplies default code for each parameter as you add each one to the pattern. You can modify this code except where comments indicate that modifications will render the pattern incompatible. The pattern author would usually add code to the expansion methods and, if applicable, to the update methods.

## Step 3: Define the Pattern

- Add template parameters:
  - Name
  - Class Name
  - Short Description
  - Type
  - Allowable Subtypes
  - Multiplicity




27

IBM

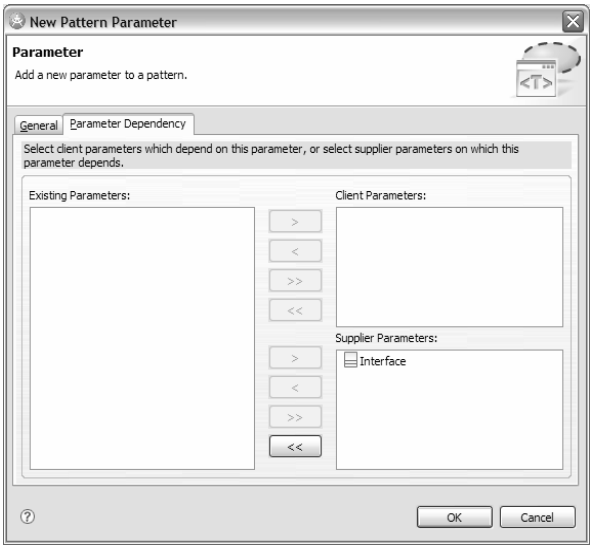
Template parameters are similar to operation parameters; they are place holders for the actual argument values that will be supplied later. For templates, this assignment by the pattern user at apply-time, of argument values to template parameters is called binding.

The principal tasks of a pattern author when creating a new template parameter are: (1) to specify its properties, such as its name, type, and multiplicity; and (2) to define its apply-time behavior, in other words, what happens when an argument value is bound to it.

# Step 3: Define the Pattern (cont.)

## Specify dependencies

Template dependencies are values bound to one template parameter that affect the apply-time behavior of other template parameters




28

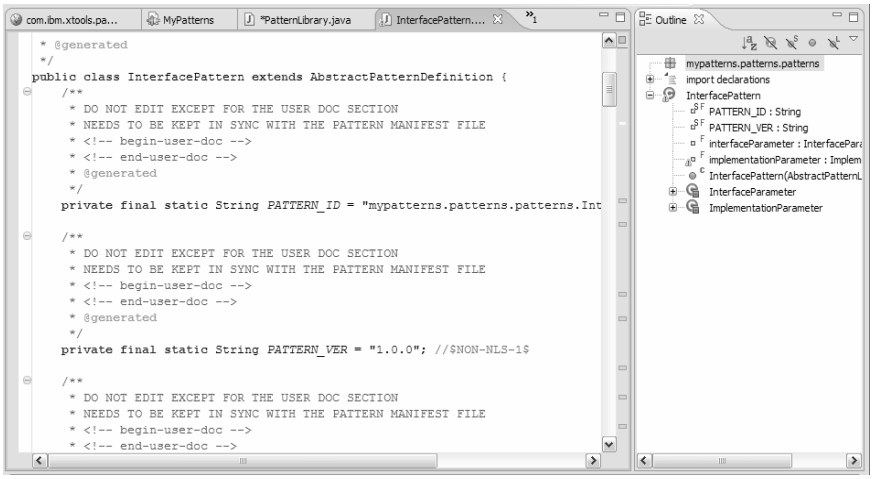
IBM

It is frequently the case that the values bound to one template parameter will affect the apply-time behavior of other template parameters. These relationships are called template parameter dependencies and provide direct support for them in the Patterns Framework and pattern authoring tools. Hot spot methods of a dependent parameter are called by the framework when the parameter that they are dependent upon has its binding modified, such as when a user assigns a new argument value.

For example, in the Interface pattern, every time an additional interface value is bound, you want to add another implementation relationship to the class that is bound to the implementation parameter. You could do this processing with the Interface parameter's hot spot method, however, it is preferable to do this processing within one of the dependent parameter's hot spots because, in general, there might be multiple dependent parameters and dependency relationships.

## Step 4: Implement Pattern-Specific Behavior

- Implement Pattern-Specific Behavior using “hotspots”
- Hotspots are places in the pattern code where the pattern author can customize the pattern’s apply-time behavior
- Two hotspots, among many in the API, are of particular interest:
  - Expansion Methods
  - Update Methods




29

IBM

When you add a parameter to a pattern in the Pattern Authoring view, a parameter class with two expansion methods is added to the pattern implementation model.

The pattern framework creates and positions expansion methods and optional update methods in the code, known as hot spots. You add Java code to the **hot spots** to dictate the pattern behavior when an argument value is added to, or removed from, a parameter. Hot spots are clearly annotated in the default Java code with `TODO` comments. The default code is marked with Javadoc `@generated` tags. The `@generated` tags must be removed if the default code is modified, or the modifications will be overwritten when the implementation model is regenerated.

These types of hot spots are discussed further on the following slides.

## Expansion Methods

- Expand methods are called when a parameter is added to, or removed from, a pattern
  - expand(PatternParameterValue value)
    - Add pattern behavior after an argument value is supplied to a template parameter.
  - expand(PatternParamterValue.Removed value)
    - Allows you to add behavior when a user deletes a value.

30

When you add a parameter to a pattern in the Pattern Authoring view, a parameter class with two expansion methods is added to the pattern implementation model. The expansion methods are called whenever a parameter is added or removed from a pattern.

Expansion methods:

- `expand(PatternParameterValue value)`: Allows you to add pattern behavior after an argument value is supplied to a template parameter. When writing the expansion code, you should consider the effects of partial or incremental expansion.
- `expand(PatternParamterValue.Removed value)`: Allows you to add behavior when a user deletes a value.

## Update Methods

- Update Methods are used in cases where dependencies between pattern parameters have been identified
  - update(PatternParameterValue value, PatternParameterValue dependencyValue)
    - Execute behavior when two parameters with a dependency relationship have been bound to the pattern
  - update(PatternParameterValue.Maintained value, PatternParameterValue.Removed dependencyValue)
    - Called when the user removes a dependent parameter that had already been added to the pattern
  - update(PatternParameterValue.Removed value,PatternParameterValue.Maintained dependencyValue)

31

IBM

A pattern can contain one or more parameters where the argument for one parameter, the supplier parameter, is used to calculate the values for dependent (client) parameters. When the user specifies a valid argument for a supplier parameter, the update methods are called to recompute the dependent client parameters.

You add a dependency relationship by using the New Pattern wizard when adding a new pattern, or by using the Properties view from the Pattern Authoring view. When added, the code for update hot spots is added to the pattern implementation model.

The three types of update methods are generated only for the client parameter. update methods are invoked for the following activities: the addition of a supplier argument, the removal of a supplier argument, and no change to the supplier argument when the pattern is reapplied.

Using update methods, you can suppress total or partial expansion of the client parameter until the required values are specified in the supplier parameters.

`update(PatternParameterValue value, PatternParameterValue dependencyValue):` If, when creating your pattern, you identify that there is a dependency between the parameters, you will need a way to execute behavior when both of the parameters have been identified. This hot spot is called after both the parameters in the dependency have been bound to the pattern.

`update(PatternParameterValue.Maintained value, PatternParameterValue.Removed dependencyValue):` This method is called when the user removes a dependent parameter that had already been added to the pattern. In this form of the method, it is indicating that the dependency has been removed.

# Hot Spots Revisited

- *Expand()*
  - parameter is bound
- *Expand(remove)*
  - parameter is unbound
- *Update()*
  - dependency is created between arguments
- *Update(maintained, removed)*
  - Second parameter was unbound
  - Need to reconcile first parameter



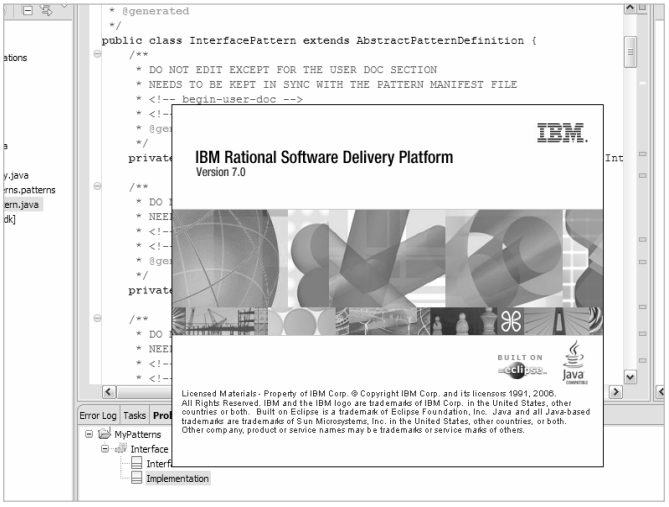
# Implement Hot Spots

- Now we just need to add behavior to the hot spots
  - Just calls to UML2, EMF, MSL APIs, right?
  - Yes, but which hot spot(s)?
- Our desired behavior
  - Create implementation relationship between class and interface
  - In class, create a method for each operation in interface
- Let's not worry about the remove hot spots right now
- So let's map behavior to hot spots

33

# Test the Pattern

The Eclipse PDE provides support for launching a version of the Workbench that can be used for testing and debugging.

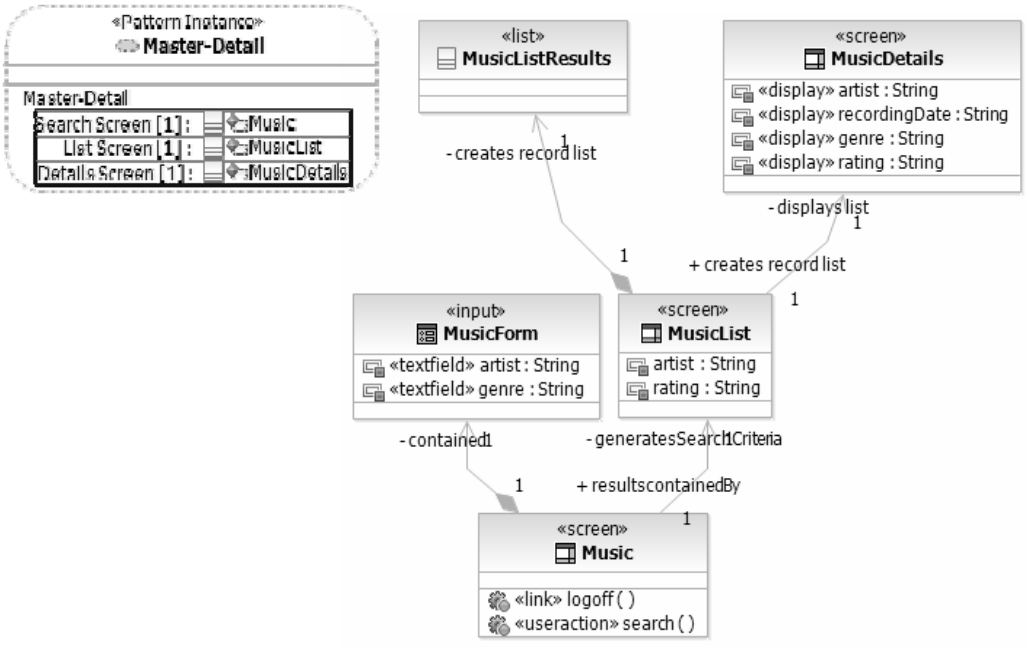



34

When working in the Plug-in Development Environment (PDE), one of the key artifacts associated with a plug-in is the `plugin.xml` file. When you open this file in the workbench, you can find out important information about the plug-in, such as dependencies, extension points, and general details about the plug-in. In addition, the workbench provides a link that will launch a new instance of the workbench with your new pattern loaded.

After the new instance has been loaded, you can test your pattern as well as work with it using the Debug perspective found in the launching instance of the workbench.

# Master Detail Pattern




35

## Demo: Create a UML Pattern

The instructor will now show you how to:

- Create a Pattern Project
- Specify dependencies between parameters

36

IBM

Watch your instructor create a simple pattern.

## Lab 12: Create the Master Detail Pattern

Complete the following tasks:

- Create the Pattern Project
- Customize Expand Methods
- Customize Update Methods
- Test the Pattern
- Extra Challenges

37

IBM

Complete Lab 12 in the student workbook.

# Review

- What are the three granular types of patterns?
- Describe examples of an architectural and a design pattern.
- Describe the role of the Pattern Framework and the Pattern Service in pattern authoring in Rational Software Architect.



IBM

## Further Information

- Rational Software Architect Help
- Web resources
- Literature


39

IBM

**Software Architect Help**

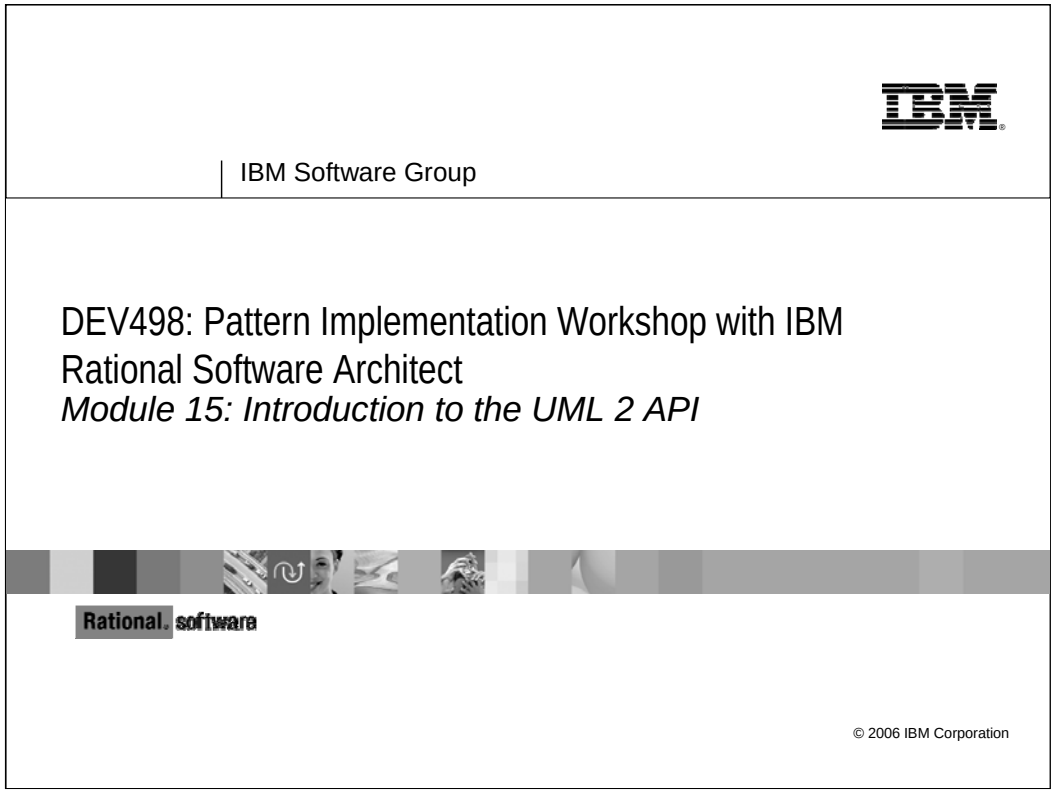
- "Authoring Patterns" Cheat Sheet
- "IBM Rational Software Modeler API"

**Web Resources**

- Martha Andrews, "Documenting your patterns using Rational Software Architect." *IBM developerWorks*. http://www-128.ibm.com/developerworks//rational/library/05/martha-andrews/
- Alan Brown and Jim Conallen, "An introduction to Model-Driven Architecture (MDA) Part II: Lessons from the design and use of an MDA toolkit." *IBM developerWorks*. http://www-106.ibm.com/developerworks/rational/library/apr05/brown/index.html
- Kenn Hussey, "Getting Started with UML2." *IBM developerWorks*. http://dev.eclipse.org/viewcvs/indextools.cgi/%7Echeckout%7E/uml2-home/docs/articles/Getting_Started_with_UML2/article.html

**Literature**

- Jim D'Anjou et al. *The Java Developer's Guide to Eclipse*. 2nd Ed. New York: Addison-Wesley, 2004.
- Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995.




## Contents

# Introduction to the UML2 API

- Objectives:
  - Describe the UML2 API
    - Profile helpers
    - Key UML API
  - Understand and use key UML2 API elements



## Where Are We?

- **Profile Helpers**
- Key UML API Elements


3

IBM

# Profile Helpers

- getStereotype()
  - For a given NamedElement, where a UML stereotype has been applied, returns the stereotype and stereotype string.

```
public static Stereotype getStereotype(NamedElement namedElement, String stereotype) {
    if(hasStereotype(namedElement, stereotype))
        return namedElement.getAppliedStereotype(stereotype);
    return null;
}
```

4

# Profile Helpers

- hasKeyword()

```
/**
 * Method determines if a user defined uml element has the indicated keyword applied.
 *
 * @param obj - java.lang.Object (should obviously be of type org.eclipse.uml2.Element)
 * @param stereotype - keyword string.
 * @return boolean - true is keyword is present, false if it is not
 */
public static boolean hasKeyword(Object obj, String keyword) {
    if((obj instanceof Element) && (keyword != null )) {
        return ((Element)obj).hasKeyword(keyword);
    } else
        return false;
}
```

5

# Profile Helpers

- hasStereotype()

```
/**
 * Method determines if a user defined uml element has the indicated uml stereotype applied.
 *
 * @param obj - java.lang.Object (should obviously be of type org.eclipse.uml2.Element)
 * @param stereotype - fully qualified stereotype string.
 * @return boolean - true is stereotype is applied, false if it is not
 */
public static boolean hasStereotype(Object obj, String stereotype) {
    if(obj instanceof Element && stereotype != null ) {
        Element element = (Element)obj;
        return element.getAppliedStereotype(stereotype) != null;
    } else
        return false;
}
```

6

## Where Are We?

- Profile Helpers
- **Key UML API Elements**


7

IBM

# Key UML API (1 of 7)

- isUMLModel(), isPackage(), isUMLOperation()

```
/**
 * Method determines if Object is in fact a root UML Model element
 *
 * @param obj - java.lang.Object
 * @return boolean - true if Object is a UML root Model element, otherwise false
 */
public static boolean isUMLModel(Object obj) {
    return UMLModelUtility.isUMLTypeMatchOf(obj, "Model");
}

/**
 * Method determines if a given Object is in fact a UML Package
 *
 * @param obj - java.lang.Object
 * @return boolean - true if Object is a UML Package, otherwise false
 */
public static boolean isPackage(Object obj) {
    return (obj instanceof Package);
}

/**
 * Method determines if a given Object is in fact a UML Operation
 *
 * @param obj - java.lang.Object (obviously should be of type org.eclipse.uml2.Element)
 * @return boolean - true if Object is a UML Operation, otherwise false
 */
public static boolean isUMLOperation(Object obj) {
    return (obj instanceof Operation);
}
```

8

# Key UML API (2 of 7)

- getAllNestedElements()

```
/**
 * Method recursively finds all elements of type "typeName" n levels deep owned by a given package
 *
 * @param pkg - org.eclipse.uml2.Package
 * @param typeName - String indicating UML type to be searched for
 * @param elemList - due to recursion this List should be passed in by user as a new list.
 */
public static void getAllNestedElements(Package pkg, String typeName, List elemList) {
    List ownedElements = pkg.getOwnedElements();
    for (int idx = 0; idx < ownedElements.size(); idx++) {
        Element element = (Element) ownedElements.get(idx);
        if (element.eClass().getName().equals("Package"))
            getAllNestedElements((Package) element, typeName, elemList);
        if (element.eClass().getName().equals(typeName))
            elemList.add(element);
    }
}
```

9

# Key UML API (3 of 7)

- findAllClasses(), findAllEnumerations()

```
/**
 * Method returns all Classes for a given list of uml elements, including a recursive search of packages
 *
 * @param elements - list of org.eclipse.uml2.Element
 * @return List - filtered list of org.eclipse.uml2.Class
 */
public static List findAllClasses(List elements) {
    ArrayList classes = new ArrayList();

    for (int i = 0; i < elements.size(); i++) {
        if (elements.get(i) instanceof Package) {
            Package p = (Package) elements.get(i);
            classes.addAll(findAllClasses(p.getOwnedElements()));
        } else if (elements.get(i) instanceof Class) {
            classes.add(elements.get(i));
        }
    }
    return classes;
}

/**
 * Method returns all Enumeration for a given list of uml elements, including a recursive search of packages
 *
 * @param elements - list of org.eclipse.uml2.Element
 * @return List - filtered list of org.eclipse.uml2.Enumeration
 */
public static List findAllEnumerations(List elements) {
    ArrayList enums = new ArrayList(20);

    for (int i = 0; i < elements.size(); i++) {
        if (elements.get(i) instanceof Package) {
            Package p = (Package) elements.get(i);
            enums.addAll(findAllEnumerations(p.getOwnedElements()));
        } else if (elements.get(i) instanceof Enumeration) {
            enums.add(elements.get(i));
        }
    }
    return enums;
}
```

# Key UML API (4 of 7)

- getClassByName(), getInterfaceByName()

```
/**
 * Searches a given package for an interface with the specified name.
 *
 * @param elem  the source element
 * @param name  the name of the interface
 * @return      the interface, if present. Otherwise, null
 */
public static Interface getInterfaceByName(NamedElement elem, String name) {
    return (Interface) getElementByKindWithName(elem, UMLPackage.eINSTANCE.getInterface(), name);
}

/**
 * Searches a given package for an interface with the specified name.
 *
 * @param pkg   the source package
 * @param name  the name of the interface
 * @return      the interface, if present. Otherwise, null
 */
public static Class getClassByName(NamedElement pkg, String name) {
    return (Class) getElementByKindWithName(pkg, UMLPackage.eINSTANCE.getClass_(), name);
}
```



# Key UML API (5 of 7)

- getComment()

```
/**
 * Returns the String comment for a given element
 *
 * @param element - org.eclipse.uml2.Element
 * @return String - comment for given element
 */
public String getComment(Element element) {
    List comments = element.getOwnedComments();
    for (int i = 0; i < comments.size(); i++) {
        Comment c = (Comment) comments.get(i);
        return c.getBody();
    }
    return "";
}
```

# Key UML API (6 of 7)

- getGeneralizations(), getDependencies()

```
/**
 * Check whether or not a Generalization already exists between the two
 * given Classifiers.
 *
 * @param specificTarget     the more specific target of the generalisation
 * @param generalTarget      the more general target of the generalisation
 * @return                   the generalisation, if it exists. Otherwise, null
 */
public static Generalization getGeneralization(Classifier specificTarget, Classifier generalTarget) {
    Generalization generalization = null;
    Iterator iter = specificTarget.getGeneralizations().iterator();
    while (iter.hasNext()) {
        EObject ownedElement = (EObject) iter.next();
        if (ownedElement.eClass() == UMLPackage.eINSTANCE.getGeneralization()) {
            Generalization ownedGeneralization = (Generalization) ownedElement;
            if (specificTarget.getName() == ownedGeneralization.getSpecific().getName()
                    && generalTarget.getName() == ownedGeneralization.getGeneral().getName()) {
                generalization = ownedGeneralization;
                break;
            }
        }
    }
    return generalization;

}

/**
 * Check whether or not a Dependency already exists between the two
 * given Elements.
 *
 * @param supplier           the supplier side of the dependency
 * @param source             the target side of the dependency
 * @return                   the dependency, if it exists. Otherwise, null
 */
public static Dependency getDependency(NamedElement supplierTarget, Classifier sourceTarget) {
    Dependency dep = null;
    Iterator iter = sourceTarget.getClientDependencies().iterator();
    while (iter.hasNext()) {
        EObject ownedElement = (EObject) iter.next();
        if (ownedElement.eClass() == UMLPackage.eINSTANCE.getDependency()) {
            Dependency foundDep = (Dependency) ownedElement;
            if (sourceTarget.getName() == ((NamedElement) foundDep.getSources().get(0)).getName()
                    && supplierTarget.getName() == ((NamedElement) foundDep.getSuppliers().get(0)).getName()) {
                dep = foundDep;
                break;
            }
        }
    }
    return dep;
}
```



# Key UML API (7 of 7)

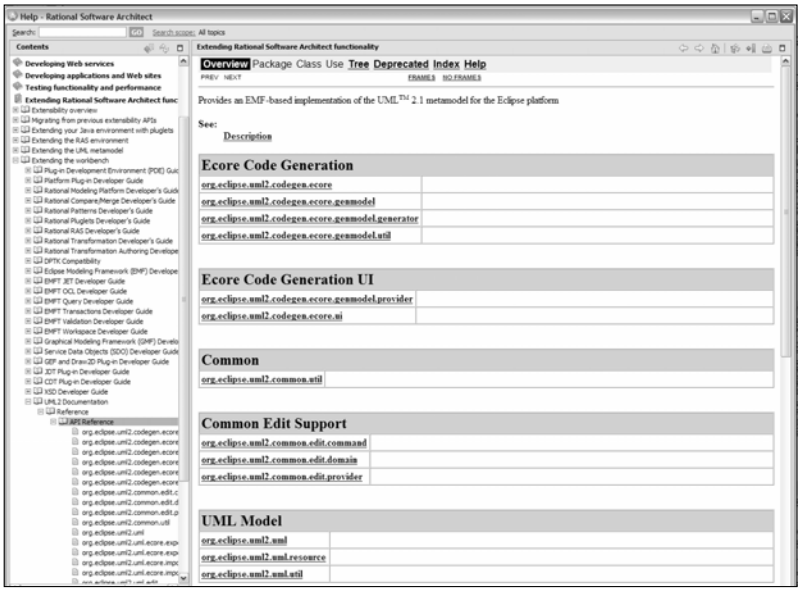
- findorCreateClass()

```
/**
 * Either find a Class with the specified name in the given package
 * or create one.
 * <p>
 * If the 'sereotypeName' is non-null then the stereotype is applied
 * to the newly created class.
 *
 * @param pkg   the package in which to look for the class
 * @param name  the name of the desired class
 * @return      the Class, created if it did not exist
 */
public static Class findOrCreateClass(Package pkg, String name, String stereotypeName) {
    Class theClass = (Class) getElementByKindWithName(pkg, UMLPackage.eINSTANCE.getClass_(), name);
    if (theClass == null) {
        theClass = (Class) pkg.createOwnedClass(name, false);
        //new api should handle name set
        //theClass.setName(name);
    }

    // Allow 'null' for non-stereotyped
    if (stereotypeName != null) {
        // Apply the stereotype if it's not already there
        Stereotype stereoType = theClass.getApplicableStereotype(stereotypeName);
        if (stereoType != null && !theClass.isStereotypeApplied(stereoType))
            theClass.applyStereotype(stereoType);
    }

    return theClass;
}
```

14

# References

- Help > Extending Rational Software Architect Functionality > Extending the workbench > UML2Documentation > Reference > API UML2Documentation




15

## Further Information

- Web resources
- Books


16

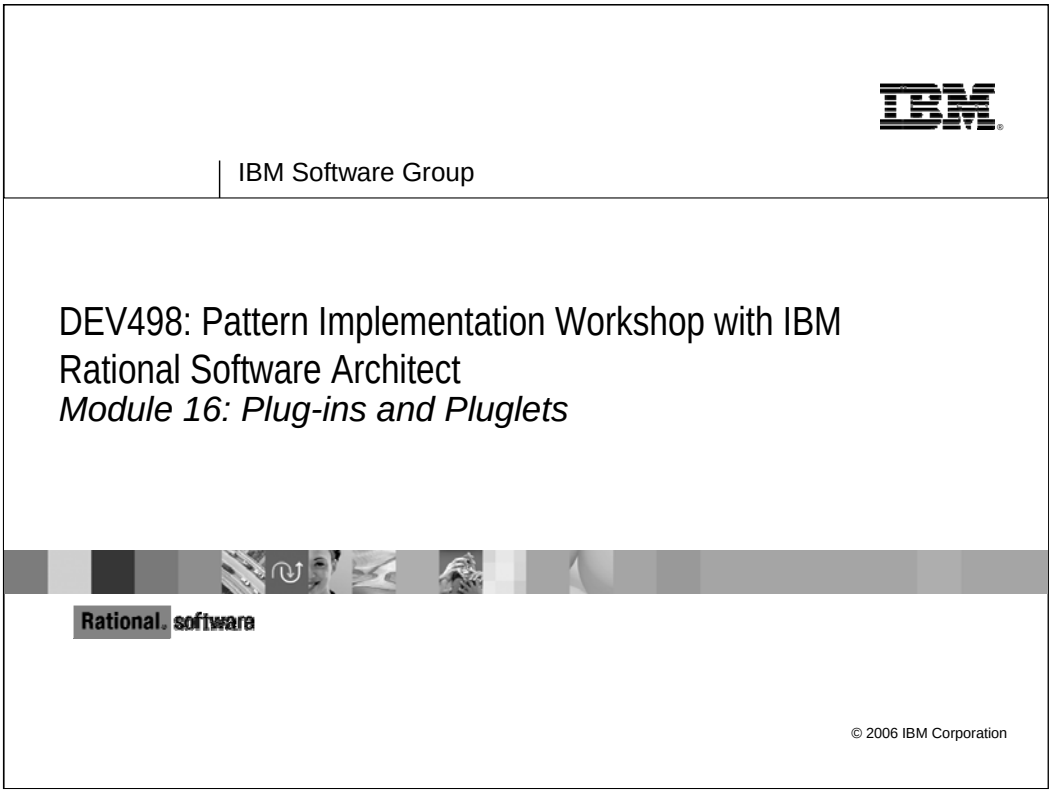
IBM

**Web Resources**

- Kenn Hussey, “Getting Started with UML2.” IBM developerWorks. http://dev.eclipse.org/viewcvs/indextools.cgi/%7Echeckout%7E/uml2-home/docs/articles/Getting_Started_with_UML2/article.html

**Books**

- Jim D'Anjou et al. The Java Developer's Guide to Eclipse. 2nd Ed. New York: Addison-Wesley, 2004.
- James Rumbaugh et al. The Unified Modeling Language Reference Manual. Boston: Addison Wesley, 2005

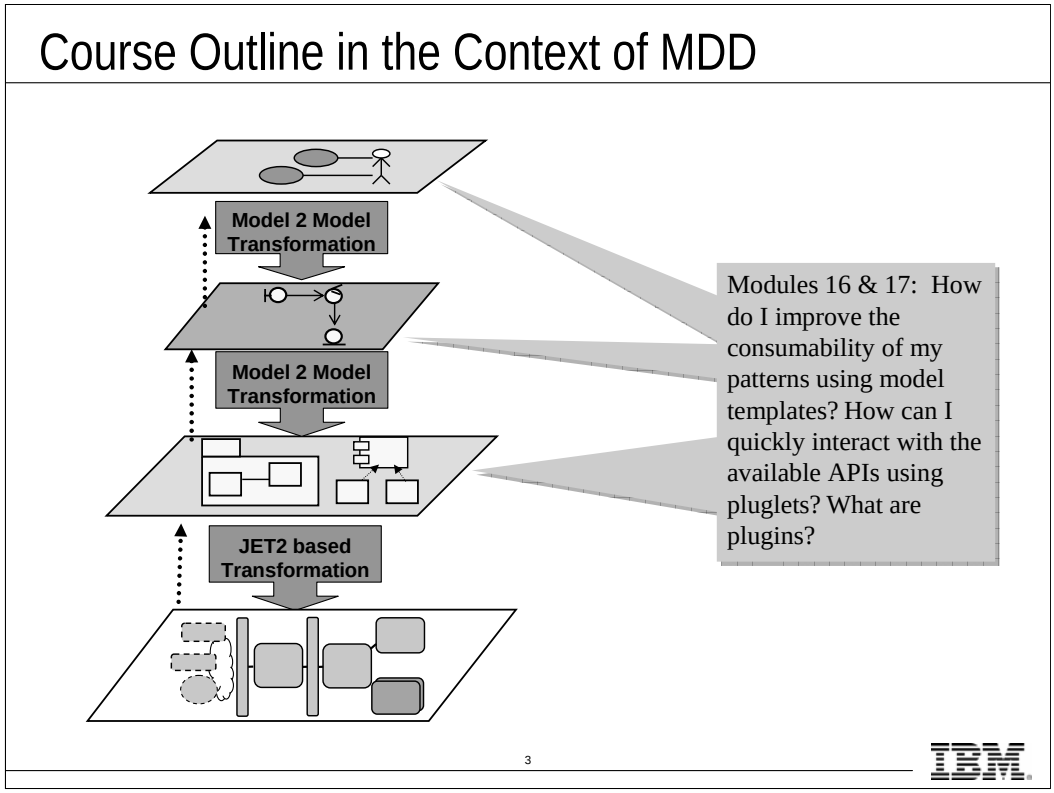



## Contents

# Plug-ins and Pluglets

- Objectives:
  - Describe the following about plug-ins and pluglets
    - The differences between them
    - The structure and contents of plug-in and pluglet projects
    - The plug-in and pluglet authoring processes
  - Create a simple pluglet






We will see this slide several times throughout the workshop. It will serve as a visual guide to the skills you are learning, and to how they fit into MDD .

# Where Are We?

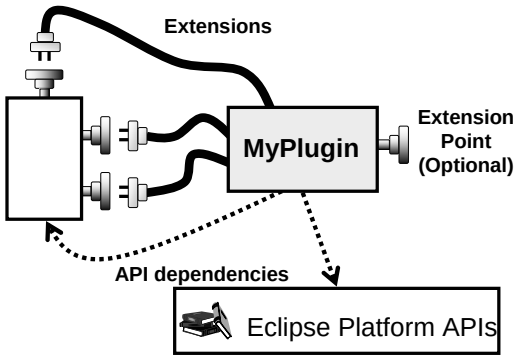
- **Plug-ins**
- Pluglets


4

IBM

## What is a Plug-in?

- A **plug-in** is a set of contributions that:
  - Provides an **extension** to the platform or another plug-in
  - Is built on specific platform or plug-in **extension points**
  - May have **dependencies** on other plug-ins or platform APIs
  - May have extension points of its own
- The platform controls and manages all contributions




5

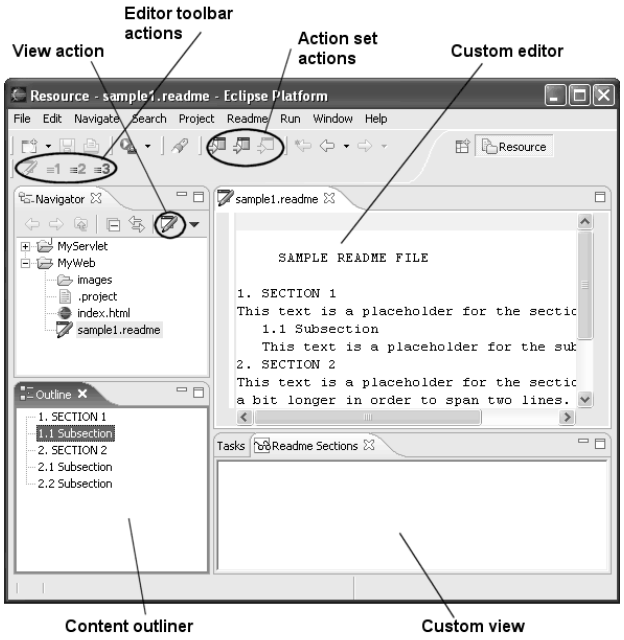
Earlier, we considered the role of plug-ins in making Eclipse a platform for application development. When you look more closely at a plug-in, you see that it has the following features:

- **Extensions:** Every plug-in contributes new behavior to the platform or to other plug-ins. This new behavior is called an extension. For example, a plug-in that contributes a simple action to a menu bar provides an extension to the platform extension point, org.eclipse.ui.actionSets.
- **Extension Points:** The extensions a plug-in provides have to be built on a specific extension point or points, which are declared on another plug-in or on the platform. Each extension point defines attributes and expected values (in an associated XML schema) that the extension's syntax must follow. Information about all available extension points is maintained in the platform's central plug-in registry.
- **Plug-in Dependencies:** A plug-in has dependencies to any other plug-ins whose code it uses, to any plug-ins it extends, and to any classes in the Eclipse platform APIs that it uses.

# Example: Contributing to the Workbench UI

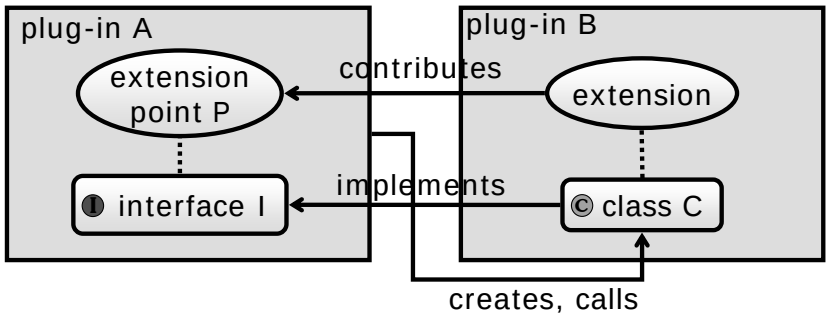
Custom extensions can be added to the Workbench UI using Eclipse plug-ins:

- Toolbar actions
- View actions
- Action set actions
- Custom editors
- Content outliners
- Custom views




6

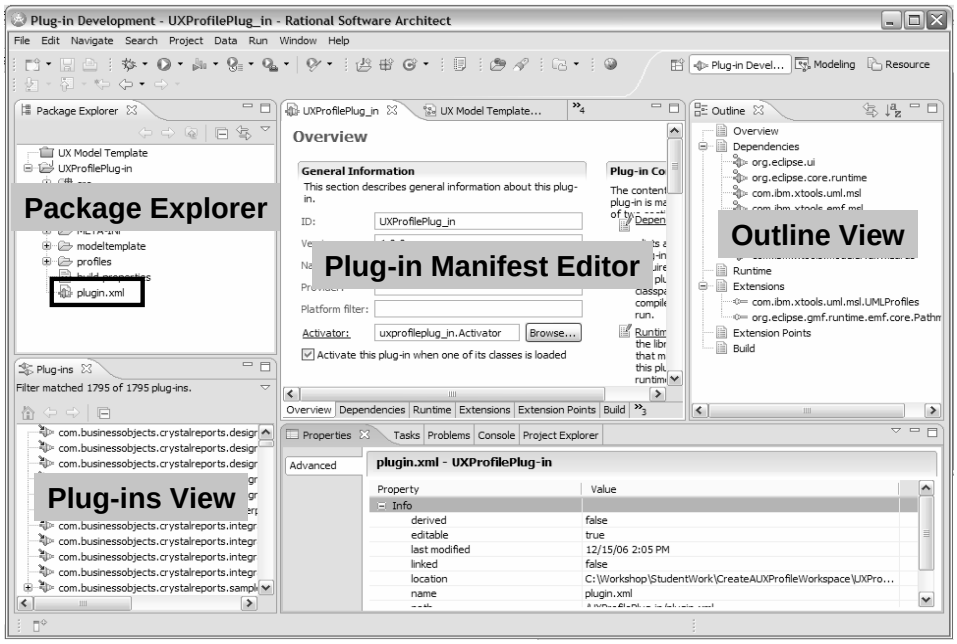
## Eclipse Plug-in Architecture – Typical Arrangement




- Plug-in design is based on the Delegation pattern:
  - Plug-in A
    - Declares extension point P and an interface I
  - Plug-in B
    - Implements interface I with its own class C
    - Contributes class C to extension point P
  - Plug-in A instantiates C and calls its I methods

7

# Plug-in Development Environment




8

The Plug-in Development Environment (PDE) assists developers with creating, developing, testing, debugging, and deploying Eclipse plug-ins. The mandate of the PDE also supports the development of fragments, features, and update sites.

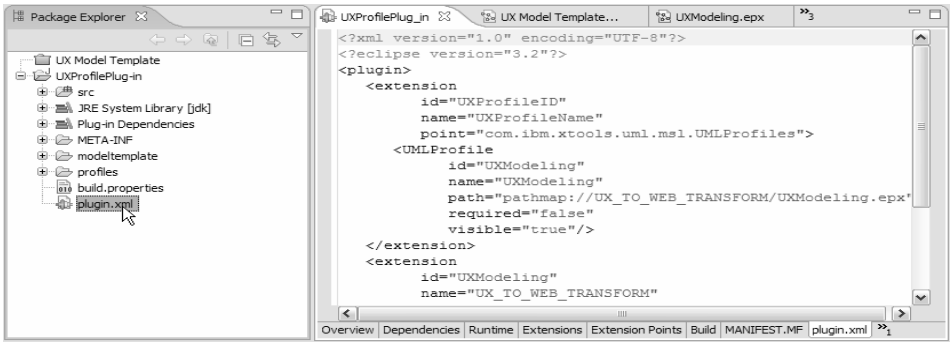
The PDE is part of the Eclipse SDK and does not have to be launched separately. In line with the general Eclipse platform philosophy, the PDE provides a wide variety of platform contributions (such as views, editors, wizards, launchers, and so on) that blend transparently with the rest of the Eclipse workbench. These contributions assist the developer in every stage of plug-in development while working inside the Eclipse workbench.

Host and Runtime Workbench Instances

When working in the PDE, the instance of the workbench in which you create the plugin is known as the host workbench. When testing and debugging the plugin, the instance of the workbench that is launched as part of the testing is known as the runtime workbench instance.

## Plug-in Project

- A plug-in includes a:
  - Manifest (plugin.xml)
    - Describes the structure, content, and dependencies of the plug-in
  - Plug-in class
    - Named Activator.java
    - Top-level Java class that represents the entire plug-in and controls class behavior at runtime.



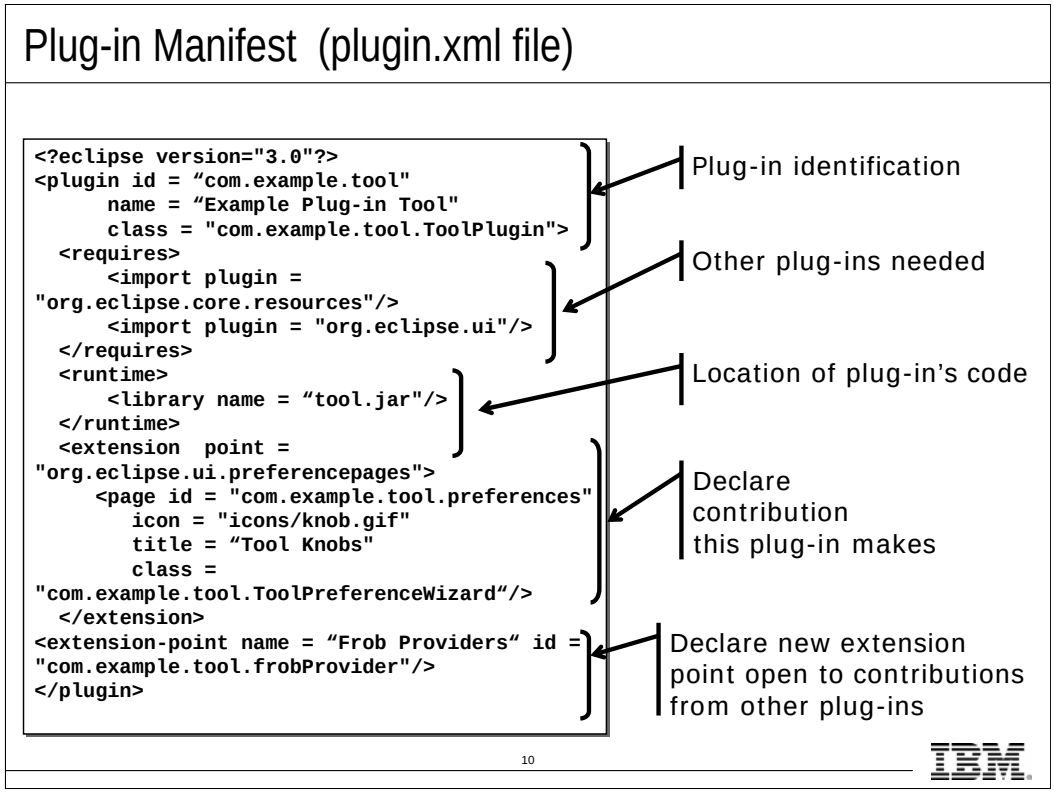

9

IBM

You develop an Eclipse plug-in in a plug-in project. A plug-in project is really just a Java project with additional packages and default items shown in the Package Explorer:

A plug-in is composed of a set of Java classes in their own namespace, and a plug-in manifest, which is an XML file that describes the contents of the plug-in. The manifest file is always called plugin.xml, and is always contained in the plug-in project's root directory. The Eclipse Platform uses manifest files to populate or update a registry of information that is used to configure the whole platform.

- **Plug-in Manifest:** The manifest includes a plug-in identifier and other meta-information, as well as sections specifying dependencies with other plug-ins, the plug-in's extensions, runtime libraries containing classes used by the plug-in, and the plug-in's extension points.
- **Source Folder:** The source folder is included with the project automatically, and includes packages containing the plug-in class (with the name PluginNamePlugin.java). The plug-in class is a top-level Java class that represents the entire plug-in, and controls class behavior at runtime. The src folder also includes code for the extensions that the plug-in class controls.
- **Build Configuration:** Created when the project is created, the build configuration is used to compile source folders into JARs. The PDE provides a simple editor for the build.properties file. The editor has form and source views. The file itself follows the Java properties format. You need to provide a number of keys and their corresponding values. Multiple values are separated using a delimiter comma.
- **Plug-in Dependencies:** Shows the parts of the Eclipse that the project uses.




An <extension-point> element has been added for this example.

<plugin>…</plugin>

The plugin element is the root element of the manifest file. The id attribute (expressed as a Java package) is the unique identifier the platform uses to reference the plug-in. The class attribute specifies the main plug-in class.

<runtime>…</runtime>

The runtime element contains a list of the libraries that contain the plug-in's implementation classes. As the project is created the New Plug-in Wizard generates this runtime element.

<requires>…</requires>

The requires element contains a list of the other plug-ins that the plug-in depends on. Each dependency is captured with the import plugin element.

<extension-point>…</extension-point>

The extension point element contains a list of extension points defined for this plug-in. The information included here is stored (and made available for developers) in Eclipse’s plug-in registry. An extension point declaration defines the id and name of the extension point, and any other plug-in specific information.

An extension point XML schema is also generated to describe the extension points, so that they can be validated and processed automatically.

<extension>…</extension>

The extension element, with the point attribute, shows which plug-in or platform extension point(s) this plug-in extends.

Note the difference between the extension-point element and the extension element. extension-point shows what extension points this plug-in offers to others. The extension element defines the functionality that extends another plug-in.

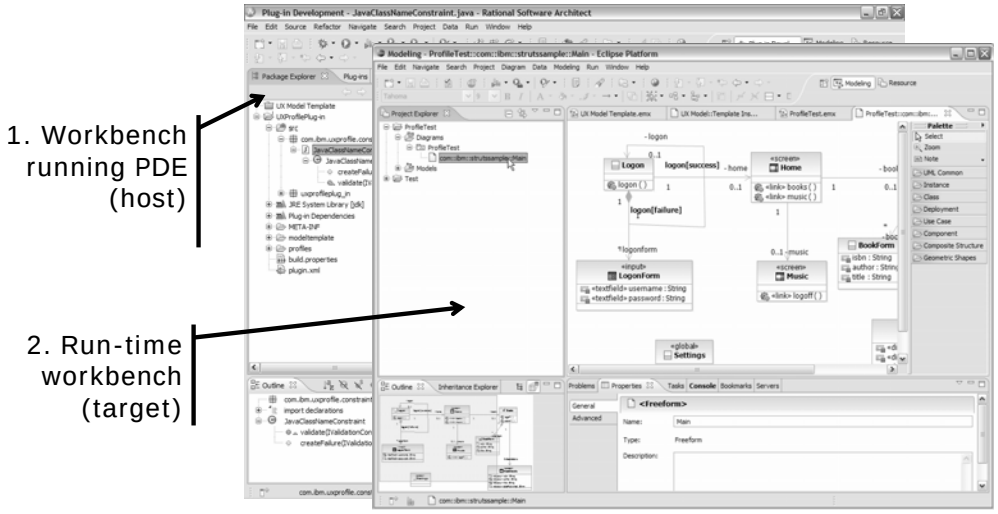
# Plug-in Activation

- Each plug-in gets its own Java class loader
  - Delegates to required plug-ins
  - Restricts class visibility to exported APIs
- Contributions processed without plug-in activation
  - Example: Menu constructed from manifest information for contributed items
- Plug-ins are activated only as needed
  - Example: Plug-in activated only when user selects its menu item
  - Initial activation starts the defined (or default) plug-in class, and then the requested function is invoked
  - Scalable for large base of installed plug-ins
  - Helps avoid long start-up times

11

# PDE Runtime and Debug Testing

- PDE launches another Eclipse workbench
- Run and debug are supported



# Running a Plug-in

- Host versus run-time instances
  - Host instance
    - Running as you develop your plug-in using the PDE and other tools
  - Run-time instance
    - Launched from Run or Debug
    - Workspace plug-ins (plug-ins under development) are merged with the External host plug-ins
    - Launch modes
      - Run 
      - Debug 

13

IBM

Eclipse JDT uses Java remote debugging. The runtime instance becomes the debug server, with the host instance as the client listening on a port for debug events.

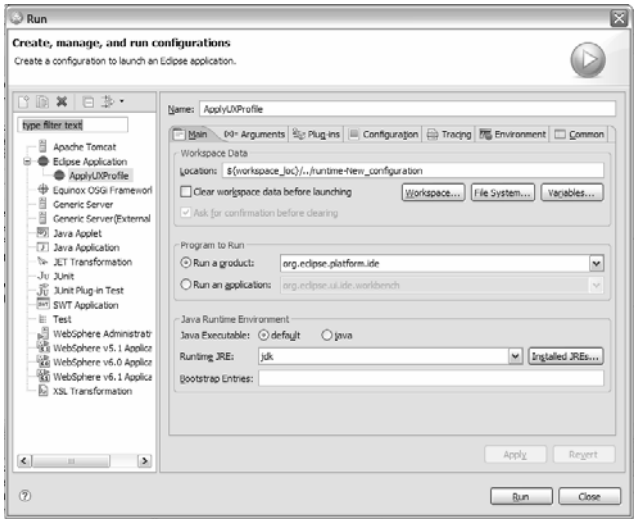
If you want to debug a standalone Java™ application and make use of the hot method replace functionality, you will need to use a Java Runtime Environment (JRE) that supports hot method replace (also called hot code replace). The installed default JRE included with the IBM® Rational® Software Development Delivery Platform provides this support.

To enable hot method replace when running with the default IBM Rational Software Development Delivery Platform JRE, go to the Arguments Tab of your Java Application launch configuration and specify -Xj9 as a JVM Argument.

When debugging Java using hot method replace, there are some limitations. To learn about these limitations, see the Java and mixed language debug limitations topic.

## Run-time Workbench Configuration Wizard

- Session Arguments
- Plug-in visibility control
- Tracing options
- Source lookup
- Environment variables
- Launcher options




14

IBM

You will use this launcher to start test sessions. You need to become comfortable with optimizing its use.

- **Arguments page:** Runtime workspace location and the ability to clear the workspace. If required, choose between the JVM, arguments, Eclipse arguments, and other launcher control details.
- **Plug-ins and Fragments:** Controls which will be included in the test session. When there are duplicates, workspace copy is used.
- **Default (1st):** is workspace plus PDE Preferences>Target Platform list
- **Features (2nd):** can test feature definitions , but requires the use of \plugins and \features directories in the workspace (see the error message when this option is selected).
- **Choose list (3rd):** Pick and choose from those in the workspace and target platform list.
- **Tracing:** Allows you to select trace control input for plug-ins that are setup to support tracing (.options)
- **Source:** Where source will be found for debugger visibility
- **Common:** Controls perspective choices after launch (overrides Preferences settings), shortcuts, and the ability to save the launch config in a file for others to use

## Where Are We?

- Plug-ins
- **Pluglets**


15

IBM

This section provides an overview of how you can extend the capabilities of Rational Software Architect by creating and using pluglets. You can also use pluglets to help create other extensibility artifacts, such as patterns and transformations.

## Pluglets Overview

- What are pluglets?
  - They are used to make minor extensions to the workbench
  - You can make pluglets that:
    - Gather model metrics (fan-in, fan-out, and model enumeration)
    - Explore APIs
  - They have available templates
  - Similar to plug-ins, they have a **pluglets.xml** manifest file and a **plugletmain()** entry point
  - They are available in Rational Software Architect, Rational Systems Designer, and Rational Software Modeler

```
<?xml version="1.0" encoding="UTF-8" ?>
- <pluglets>
  - <require>
      <import plugin="com.ibm.xtools.pluglets" />
      <import plugin="com.ibm.xtools.modeler" />
    </require>
  </pluglets>
```

Pluglet Manifest

16

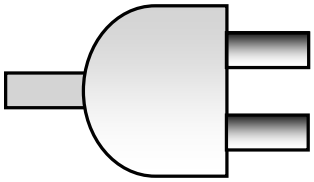
IBM

A pluglet, developed in a pluglet project, is a light-weight version of the plug-in, can provide a script-like extension to the development environment to handle routine tasks. Pluglet functionality offers you an easy way to explore and learn the application programming interfaces (APIs) offered by the workbench platform and other product extensions. From a workbench perspective your can write the statement `Platform.getWorkbench()` and gain completed access to the entire workbench and its parts. From a modeling perspective, the UMLModeler class allows you access to a model and its contents.

It is a simple Java program that runs from a top-level menu. Because pluglets have a limited scope, they are relatively easy to develop and require only minimal knowledge of how plug-ins work. As Java programs, pluglets can be developed in the Java development environment and access the workbench plug-in APIs (such as the Eclipse, UML2, EMF, and Rational Software Architect APIs). Pluglets can also be tested in the same instance of the workbench.

# Pluglet Applicability and Limitations

- Applications
  - Use for samples, one-time tools (such as a migration utility), and so on
  - Useful for exploring the extensibility APIs when building patterns and transformations
  - Very useful for obtaining access to a model in the current workspace
  - Don't use pluglets in place of workbench product extensions
  - Shipped samples should include source code
    - Required for the Ready for Rational Software program
- Limitations
  - Requires a separate Rational Software Architect session to run in the debugger
  - `System.out` is NOT written to console view. Use `Pluglet.out`
    - Start Rational Software Architect with the `–consolelog` parameter and you will see `System.out`.
    - Can be helpful if your pluglet includes other classes that require debugging

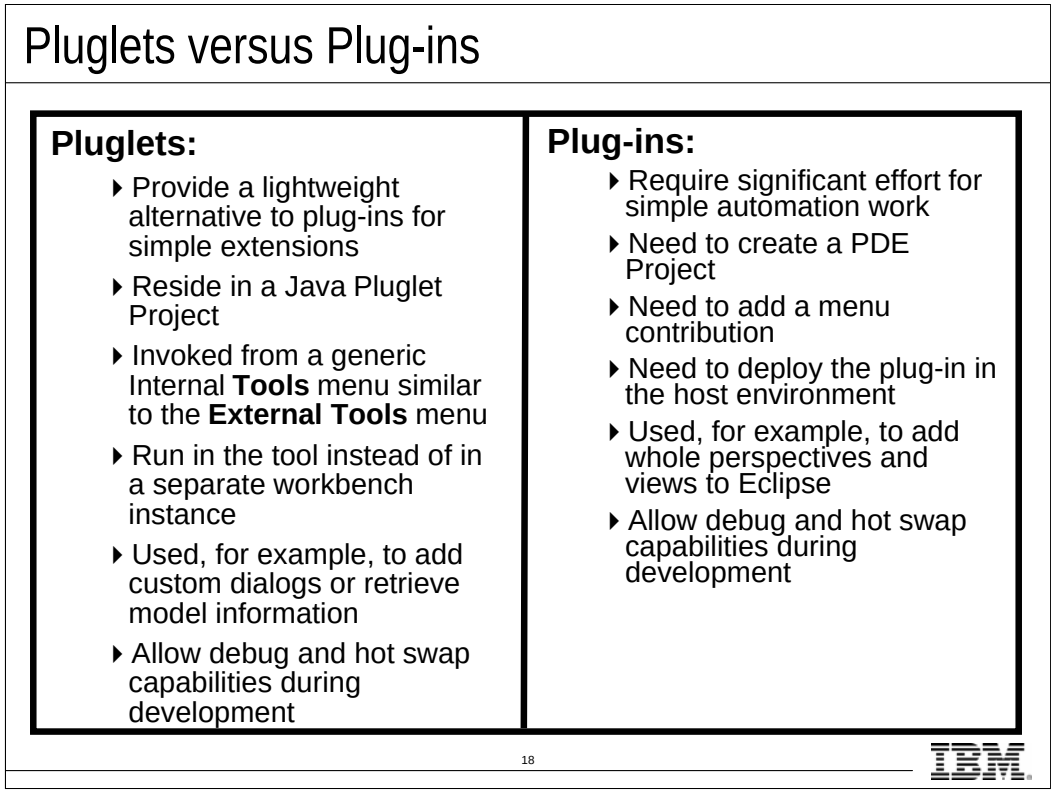


17

If you reference other classes they probably won't be extending the Pluglet class, so the out method is not available. Using System.out with the `–consolelog` startup of Rational Software Architect will allow you to see debugging output from those other classes

This is very useful for working with and exploring the api's (uml2, emf, rsa, and so on). It can be used to figure out how a pattern or transformation should work. Then the code can be put into the pattern or transformation.

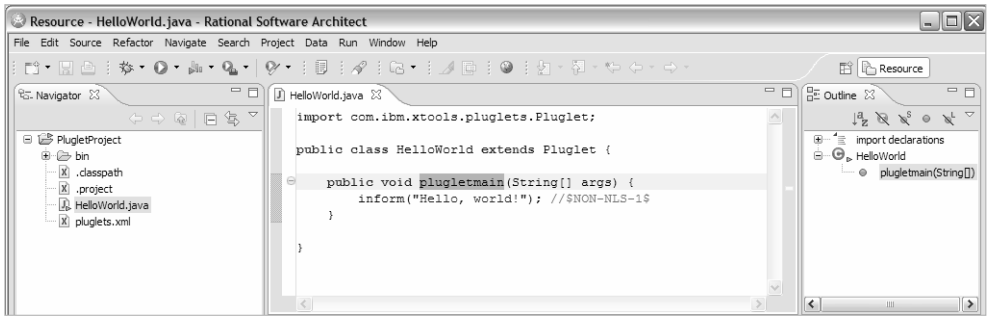
This method can save you a great deal of time, as the pluglet runs in the same eclipse instance.




You can create pluglets to handle routine tasks, and pluglet functionality offers you an easy way to explore and learn the APIs offered by the workbench platform and other product extensions. From a workbench perspective you can write the statement `Platform.getWorkbench()` and gain complete access to the entire workbench and its parts. From a modeling perspective, the UMLModeler class allows you access to a model and its contents.

# Creating Pluglets

- Create a pluglets project in the Java perspective
- Fill in the pluglets manifest
- Create Java classes
- Test in the same workbench
  - Use debug features in a separate session




19

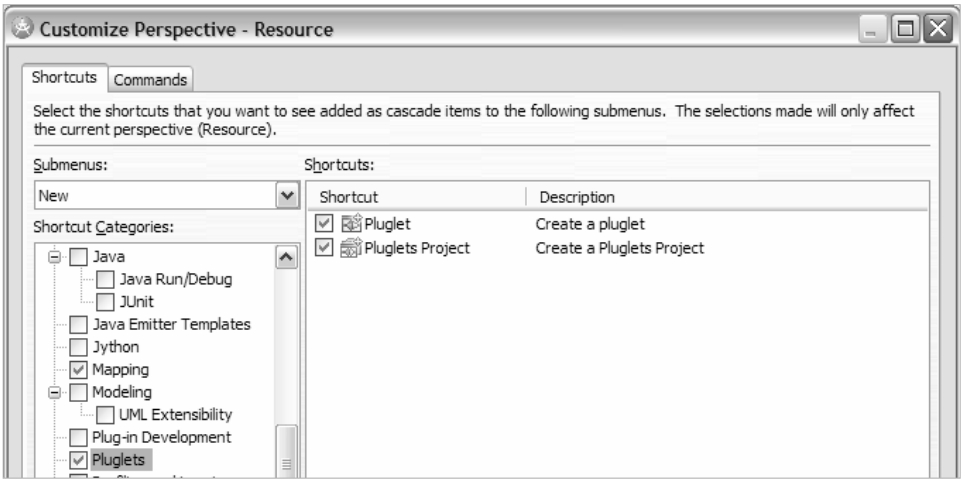
To create a pluglet, you only need the class implementing the desired behavior and the pluglet.xml file (indicating the other plug-ins to import).

You can test your pluglet directly on the same session on which you are developing it. This is as opposed to plug-ins testing, where you need to start a target workbench instance. Iif you like to debug the pluglet and use breakpoints you will need to start a new session.

1. Create a new plug-in:
   a. **Click File > New > Pluglets Project**. The New Pluglets Project wizard appears.
   b. On the first page of the New Pluglets Project wizard, enter the project name.
   c. Either accept the default directory or specify an output directory for the new pluglets project and Click **Finish**.
2. Add a pluglet to the pluglet project:
   a. Click **File > New > Pluglet** to display the new pluglet page.
   b. On the New Pluglet page, select one of the templates for the new pluglet.
   c. Click **Next**.
   d. Enter the name of the new pluglet in the **Name** field.
   e. Click **Finish** to start working with the pluglet.
3. Write pluglet code. Pluglets extend the Pluglets class, and the plugletmain method is called first.
4. Test the pluglet. Select the newly created pluglet in the Model Explorer view. Click **Run > Internal Tools > Pluglet-name**.

## Enabling Pluglet Development

To begin developing pluglets, add **Plugets** and **Pluglet Projects** to the **New** menu.




20

IBM

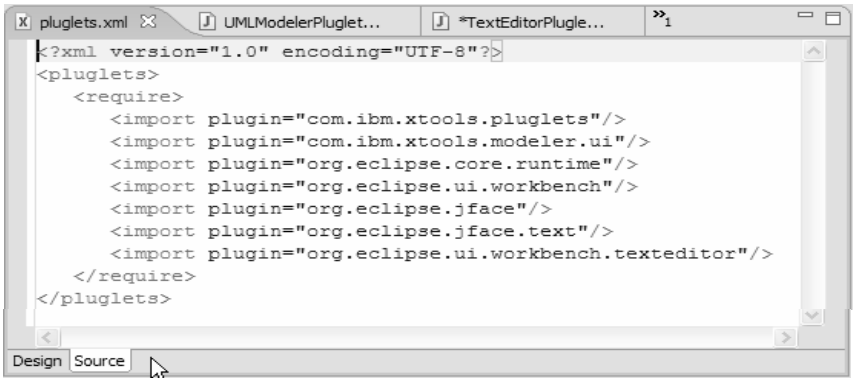
To enable pluglets development, you first need to customize the perspective to include menu items for creating pluglets and pluglets projects:

1. Click **Window > Customize Perspective**.
2. In the Customize Perspective window, click the **Shortcuts** tab and be sure **New** is specified in the **Submenus** list.
3. In the **Shortcut Categories** list, select **Pluglets** to enable the related shortcuts. Make sure that the **Pluglet** and **Pluglets Project** check boxes are checked in the **Shortcut** list.
4. Click the **Commands** tab. In the **Available command groups** list, select **Pluglets** and **Modeling**.
5. Click **OK**.

To check if your perspective is customized, click **File > New** and check if the **Pluglets Project** was added to the pull down menu.

# Pluglets Manifest (pluglets.xml)

- Used to identify dependencies on other **plug-in libraries**
  - Serves as the pluglets' classpath
  - Minimally references the `pluglets` plug-in
  - Requires manual update when new plug-in dependences occur
  - One `pluglets.xml` file per project
  - Use **Help > Extending Rational Software Architect functionality > Extensibility Reference > API Reference** for help on plug-ins to import

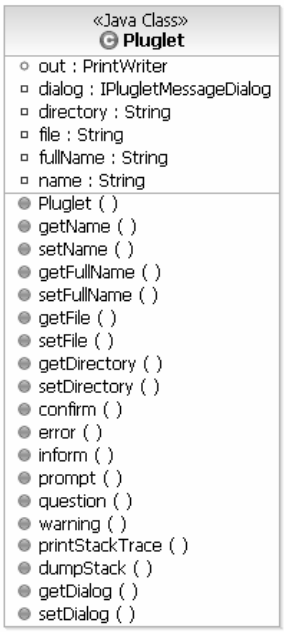



21

This effectively provides the classpath for your pluglet by identifying plug-ins whose libraries (JAR files) you have a dependency on.

## The Pluglet Class

- Provides properties and basic services used by pluglets
- **PrintWriter** provides the output for the pluglet
- The class provides convenience dialogs
  - Confirm
  - Prompt
  - Error
  - Question
  - Warning




22

To use the basic pluglet properties and services, pluglet must extend the Pluglet class from the Eclipse pluglet API. This class includes, for example, the following methods:

- `getName`: retrieve the name of the pluglet.
- `getDirectory`: retrieves the full path of the pluglet directory.

Use of this class and extending the class is optional.

# Development Considerations

- **Pluglet** class
  - All pluglets extend this class
  - Contains many helper methods for user interaction, basic i/o, pluglet info, and so on

| Field/Method | Description |
|---|---|
| `out` | `Printwriter` field. Use in place of `System.out` |
| `inform,question`<br>`prompt,confirm,`<br>`warning, error,` | User interaction dialogs |
| `dumpstack,`<br>`printStackTrace` | Diagnostics |
| `getXXX, setXXX` | get/set pluglet data.<br>directory, file, pluglet name |

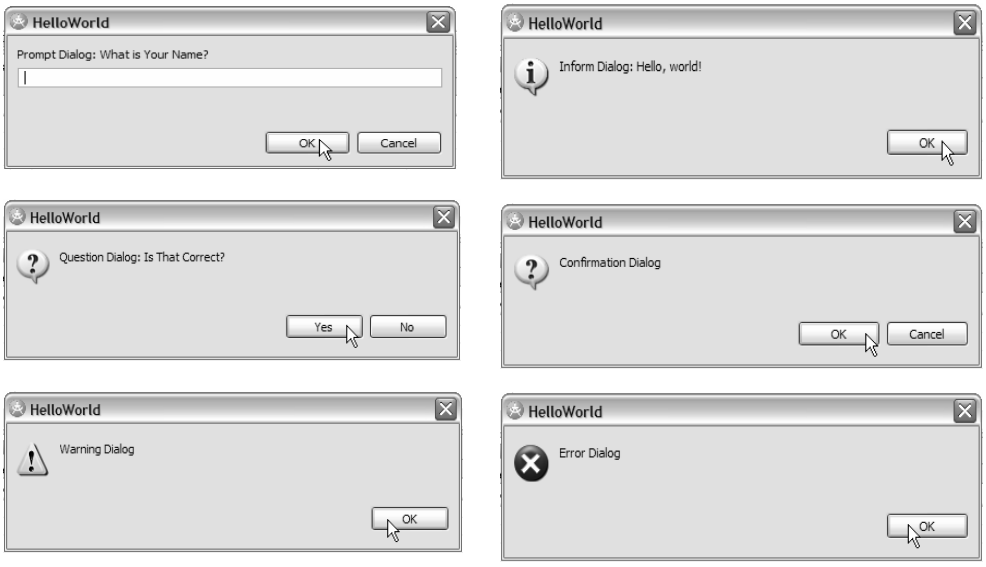
23

Examples of user interaction dialogs are on the next slide.

com.ibm.xtools pluglets:

- Class Pluglet
- java.lang.Object com.ibm.xtools.pluglets.Pluglet
- Direct Known Subclasses:
- InsertDateAndTime, ListPerspectives, ListProjects, ShowSelection
- public class Pluglet extends Object
- Provides pluglet properties and basic services used by pluglets. To use these properties and basic services, a pluglet class must extend this class. Use of this class (and thus extending this class) is optional.
- Field Summary PrintWriterout
  The output for this pluglet. Constructor SummaryPluglet()
  Method Summary:
  - `booleanconfrim(String message)`: Displays a message dialog with **OK** and **Cancel** buttons.
  - `Booleanconfirm(String message,string title`: Displays a message dialog with OK and Cancel buttons and the given title.
  - `VoiddumpStack():` Prints a stack trace of the current thread to the pluglet output writer.
  - `voiddumpStack(PrintWriter writer)`: Prints a stack trace of the current thread to the specified print writer.
  - `Voiderror(String message)`: Displays an error dialog with an O**K** button.
  - `Voiderror(String message, String title)`: Displays an error dialog with an OK button and the given title.
  - IPlugletMessageDialoggetDialog`()`: Retrieves the host's IPlugletMessageDialog implementation.
  - `StringgetDirector()`: Retrieves the full path of pluglet directory.

## Example Pluglet Class Dialogs




More sophisticated dialogs are possible using the Eclipse JFace and SWT Dialog classes.

24

IBM

- `StringgetFile()`: Retrieves the full path of the pluglet file.
- `StringgetFullName()`: Retrieves the full name of the pluglet.
- `StringgetName()`: Retrieves the name of the pluglet.
- `voidinform(String message)`: Displays a dialog with an **OK** button.
- `voidinform(String message, String title)`: Shows an **OK** button and title.
- `voidprintStackTrace(Throwable t)`: Prints the throwable and its backtrace to the pluglet output writer.
- `voidprintStackTrace(Throwable t, PrintWriter writer)` Prints the throwable and its backtrace to the specified print writer.
- `Stringprompt(String message)`: Displays an input dialog with a prompt message, a text input field, and **OK** and **Cancel** buttons.
- `Stringprompt(String message, String initialText)`: Displays an input dialog with a prompt, the text input field initial text, and **OK** button and **Cancel** buttons.
- `Stringprompt(String message, String initialText, String title)`: Displays an input dialog with a prompt message, the text input field initial text, the title, and **OK** button and **Cancel** buttons.
- `booleanquestion(String message)`: Displays a dialog with **Yes** and **No**.
- `booleanquestion(String message, String title)`: Displays a question message dialog with **Yes** and **No** buttons and the given title.
- `voidsetDialog(IPlugletMessageDialog dialog)`: Sets the host's IPlugletMessageDialog implementation.
- `voidsetDirectory(String directory)`: Sets the full path of pluglet directory.
- `voidsetFile(String file)`: Sets the full path of pluglet file.
- `voidsetFullName(String fullName)`: Sets the full name of the pluglet.
- `voidsetName(String name)`: Sets the name of the pluglet.
- `voidwarning(String message)`: Displays a warning dialog with an **OK** button.
- `voidwarning(String message, String title)`: Displays a dialog with an **OK** button and title.

# Some Pluglet API Entry Points

Rational Software Architect and Eclipse or JDT classes with static methods that open up particular workbench API's

| Class | Plug-in | Description |
|---|---|---|
| `PlatformUI.getWorkbench` | org.eclipse.ui | Central access point for workbench UI |
| `ResourcesPlugin.getWorkspace` | org.eclipse.core.resources | Workspace |
| `UMLModeler` | com.ibm.xtools.modeler.ui | Central access point for UML model access |
| `UMLDiagramResourceUtil` | com.ibm.xtools.viz.ui | Central access point UML diagrams |
| `JavaCore` | org.eclipse.jdt.core | Central access point for Eclipse JDT model |
| `JavaUI` | org.eclipse.jdt.ui | Central access point for Java UI |

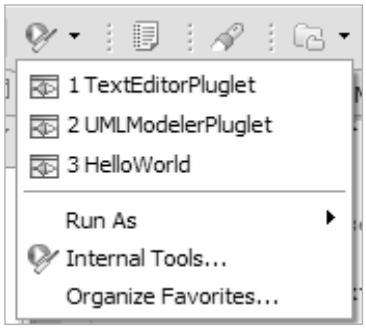
25

These are classes in Eclipse, Java, and Rational Software Architect that contain static methods that provide access to root objects in the class hierarchy of these components. They provide entry points for pluglets.

# Testing the pluglet

- Context menu from selected pluglet
  **Run > Pluglet**


- Or from the toolbar
  - Including Modeling perspective




26

## Demo: Create a Pluglet

The instructor will now show you how to:

- Enable Pluglet sub-items in the **New** menu
- Create a pluglet
- Run the pluglet

27

IBM

Watch your instructor create a simple pluglet.

## Lab 13: Create a Pluglet

- Given:
  - Pluglet project, **PlugletProject.zip**
  - Code fragments
- Complete the following tasks:
  - Create the Workspace
  - Configure the Perspective
  - Import the Pluglet
  - Complete the Pluglet
  - Run the Pluglet
  - Export the Pluglet

28

IBM

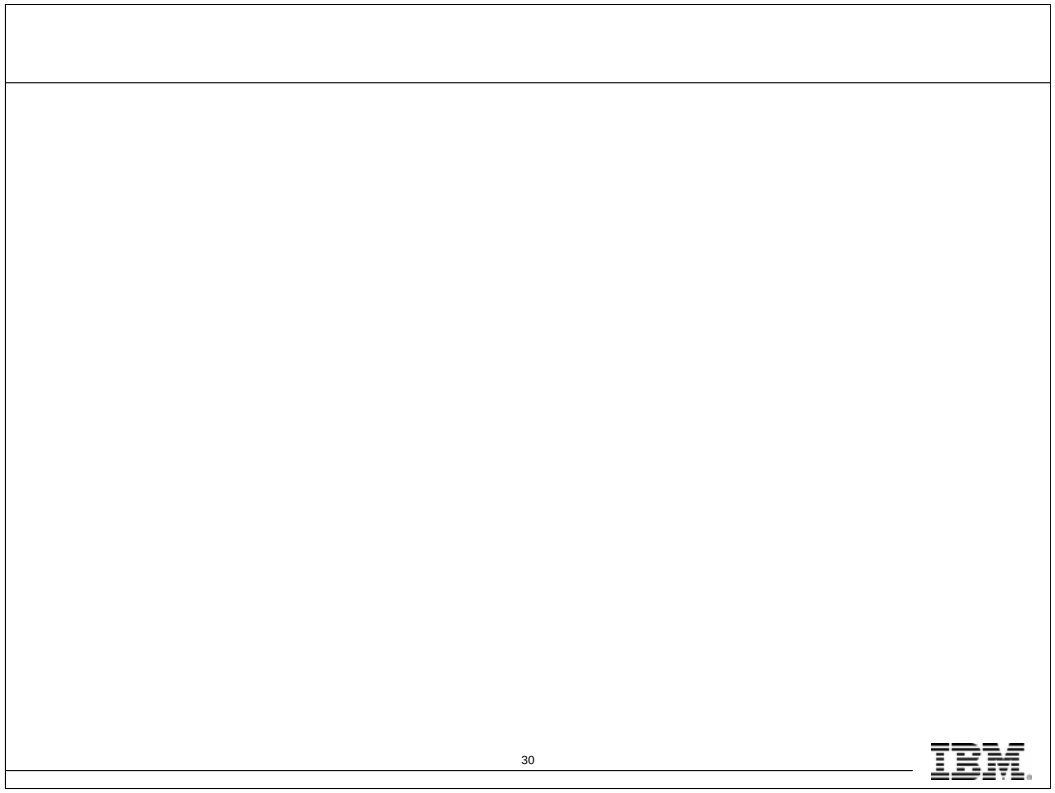
Complete Lab 13 in the student workbook.

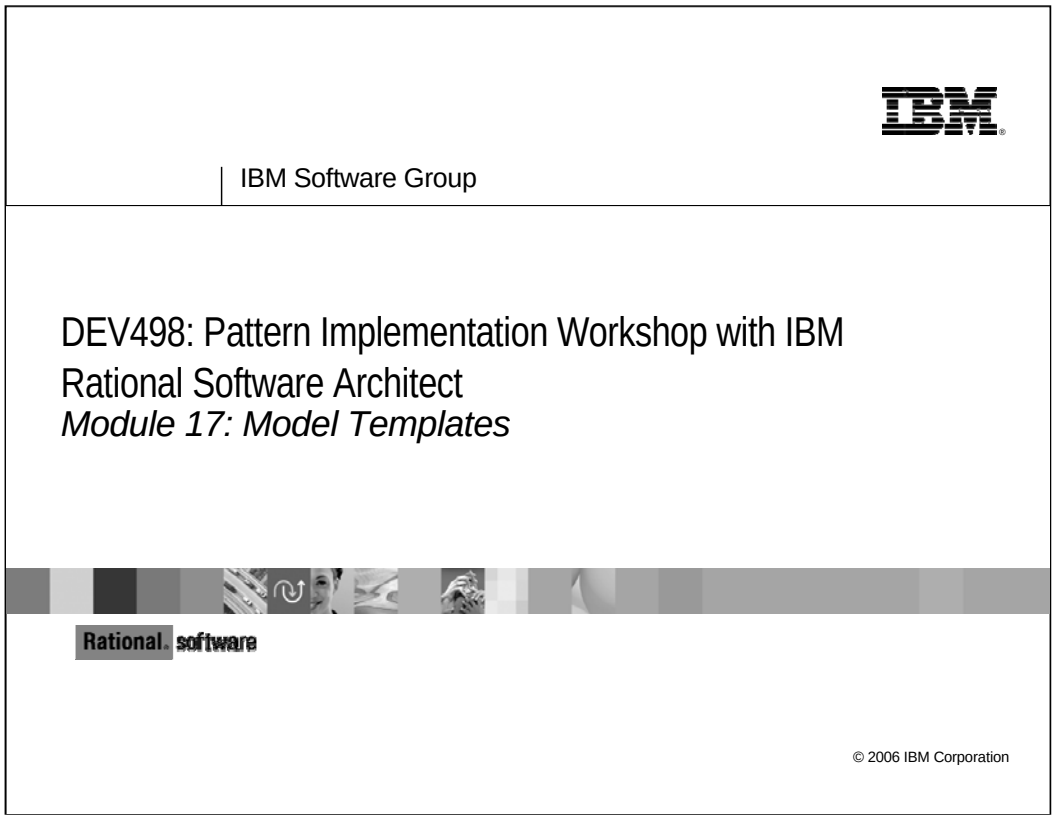
## Review

- What is the difference between a host and run-time workbench?
- What are the components of a plug-in project?
- For what purposes can pluglets be used?

29

IBM






## Contents

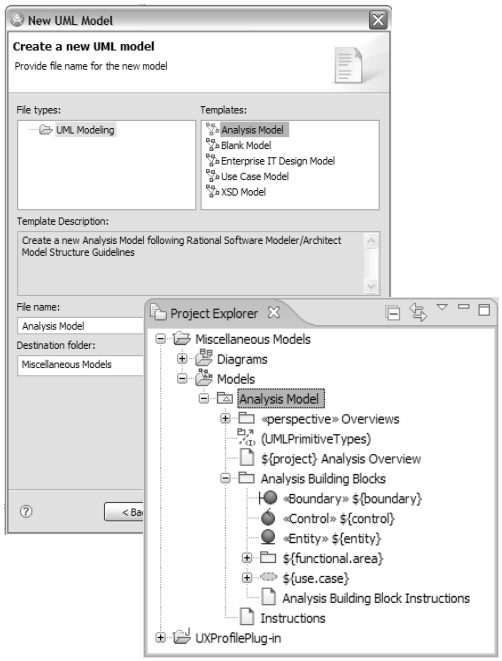
# Model Templates

- Objectives:
  - Describe Rational Software Architect model templates
    - The uses for model templates
    - How to design a model template
    - How to create a model template in Rational Software Architect
  - Describe the relationship between model templates, profiles, UML patterns, and transformations

2

# Model Templates

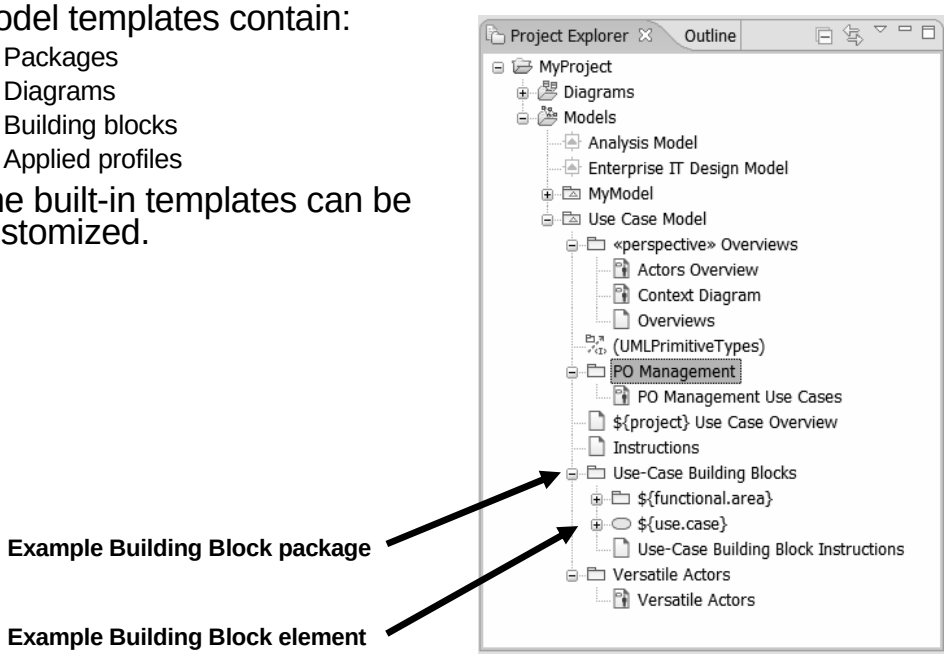
- Allow the user to create a new model based on a pre-existing structure
  - Built-in templates available from the New Project wizard
- Used in conjunction with:
  - **Profiles:** Guide the user in structuring the model as the use profile stereotypes
  - **UML Patterns:** Used to populate the model with standard elements and structures, such as model elements, package structures and diagrams
  - **Transformations:** Provide a standard, structure, source , or target for a custom transformation




3

## Review: Populating a Model Template

- Model templates contain:
  - Packages
  - Diagrams
  - Building blocks
  - Applied profiles
- The built-in templates can be customized.




4

Every template contains a «modelLibrary» package called TemplateName Building Blocks. This package contains chunks of model content that you can use to build the design model more quickly. Building blocks act as template model elements. You can copy (CTRL+C) and paste (CTRL+V) the building block elements to create new elements for your model.

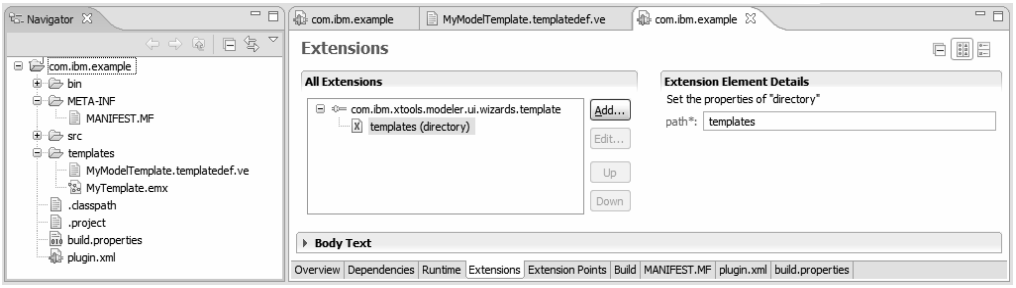
To use a building block element:

- In the Project Explorer, copy a building block element from the building blocks package and paste it in the desired location in the model.
- Right-click the new element and choose Find/Replace to change the placeholder name {$name} to the desired name.

A best practice for naming diagrams is to come up with a descriptive name and then add the diagram type. For example, the use-case diagram above is called “PO Management Use Cases.” You might also call it “PO Management Use-Case Diagram.”

# Creating a Model Template

- Build a new model structure based on a blank model or other model
- Create a Plug-in Project
- Add .emx file to the Plug-in
- Connect to an extension point
- Specify template details in the .ve file




5

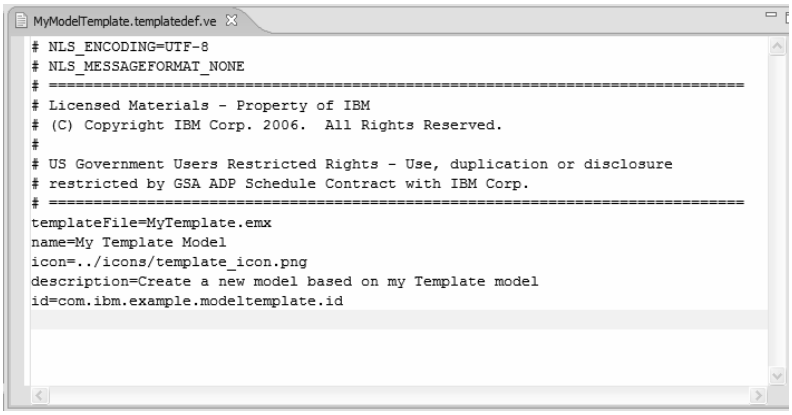
You can save a model in the workspace as a model template. Model templates can contain pre-defined model elements and provide the basic structure of a new model. Model templates can be distributed to other team members to ensure that there is a consistent model format within a project.

To export a model as a template:

1. Build a model that represents the structure that you wish to make available to others for reuse.
2. Create a Plug-in project.
3. Copy the .emx file for the model into the plug-in project. By convention, you should place the .emx file into a folder names templates.
4. Connect to the com.ibm.xtools.modeler.wizards.template extension point. Add a directory element and point to the location where you placed the template file.
5. Create a new text document with a .ve extension.
6. Add details to the .ve file that describe your template.
7. Test.
8. Deploy.

# Specify Template Details

- The .ve file allows you to specify details for your model template, including:
  - Name
  - Description
  - Icon




6

The .ve file is used to specify details that will make your model template more consumable by the template's end users. Within the .ve file you can specify the name, description, and an icon for the template. To ensure that this information is shown in support of just your template, you also specify the name of the profile file, as well as the id of the plug-in that contains the template.

## Lab 14: Create a UX Model Template

- Complete the following tasks:
  - Create the Model Template
  - Add the Model Template to a Plug-in
  - Apply a Profile to the Model Template

7

IBM

Complete Lab 14 in the student workbook.

## Review

- What elements can be found within a model template?
- How are model templates distributed?
- What roles can model templates play alongside other artifacts?

8

IBM

## Further Information

- Web resources
- Literature


9

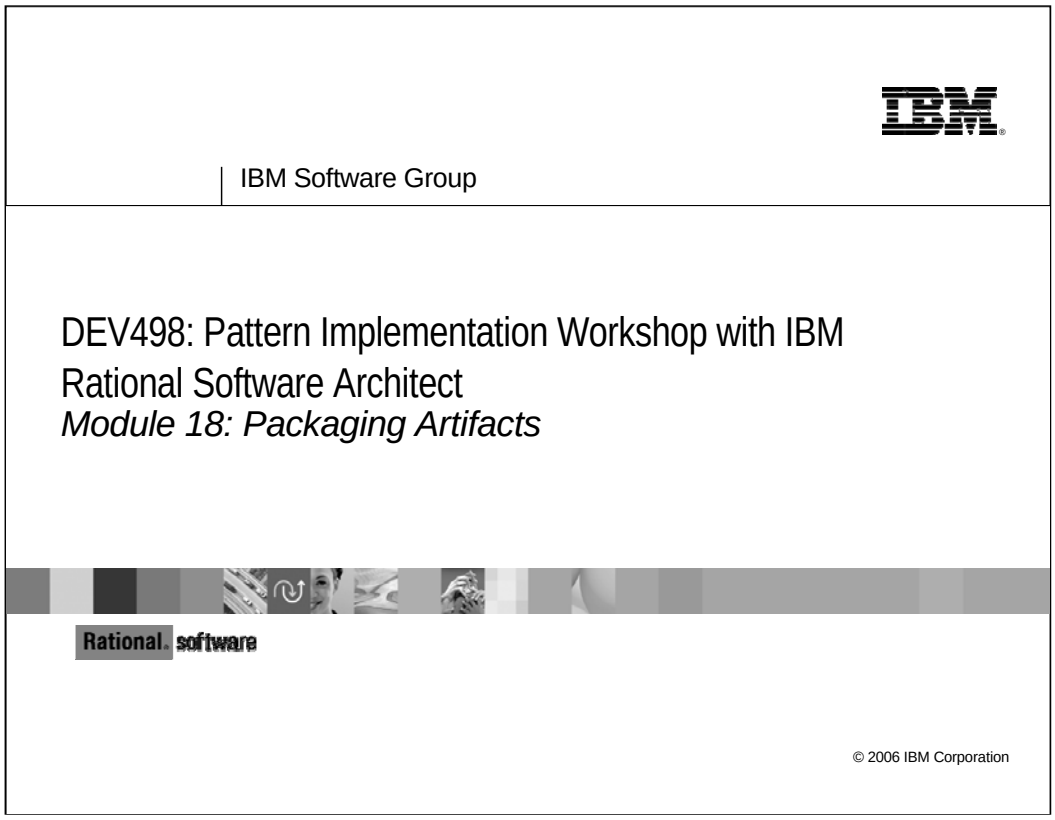
IBM

**Web Resources**

- Bran Selic. “Unified Modeling Language version 2.0.” *IBM developerWorks*, http://www-128.ibm.com/developerworks/rational/library/05/321_uml/

**Literature**

- James Rumbaugh et al. *The Unified Modeling Language Reference Manual.* Boston: Addison Wesley, 2005.

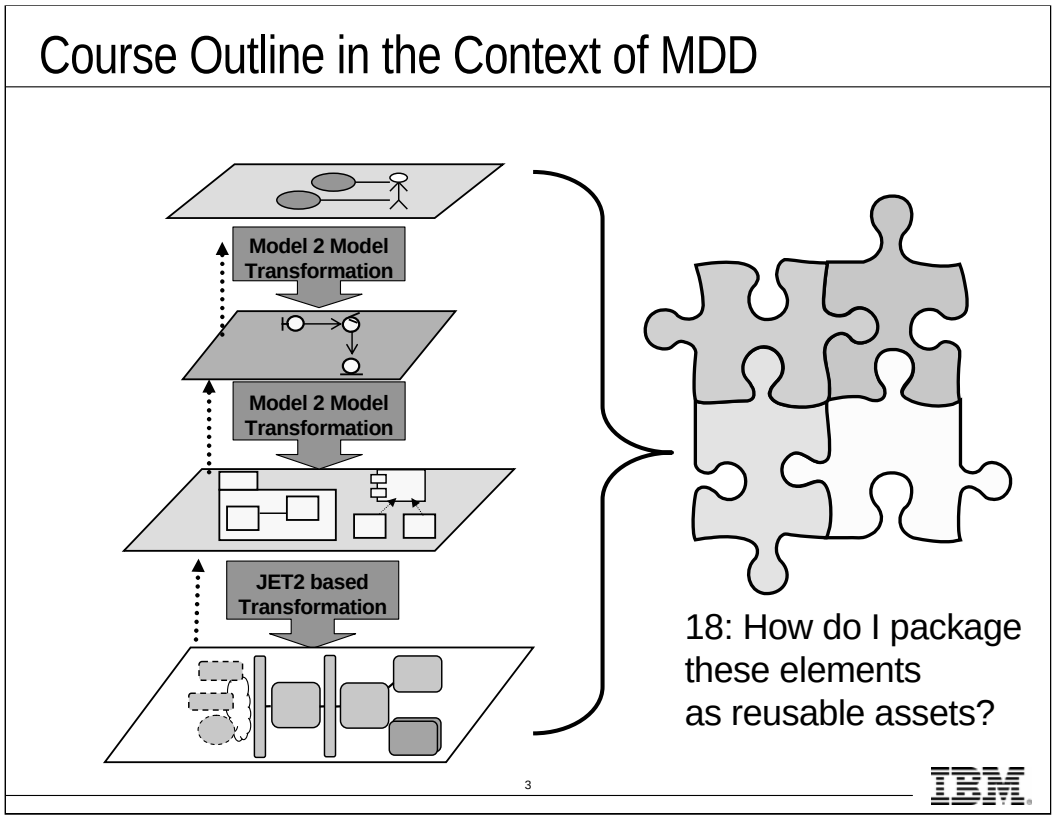



## Contents

# Packaging Artifacts

- Objectives:
  - Describe the methods of managing and packaging extensibility artifacts:
    - Plug-ins
    - Features
    - Reusable Asset Specification (RAS) Archives
  - Create a RAS archive

2




We will see this slide several times throughout the workshop. It will serve as a visual guide to the skills you are learning, and to how they fit into MDD .

## Where Are We?

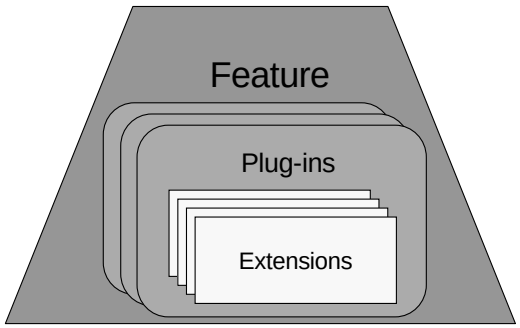
- **Eclipse Features**
- Reusable Asset Specification Archives


4

# Eclipse Features: Packaging and Installing Plug-ins

## Eclipse Features

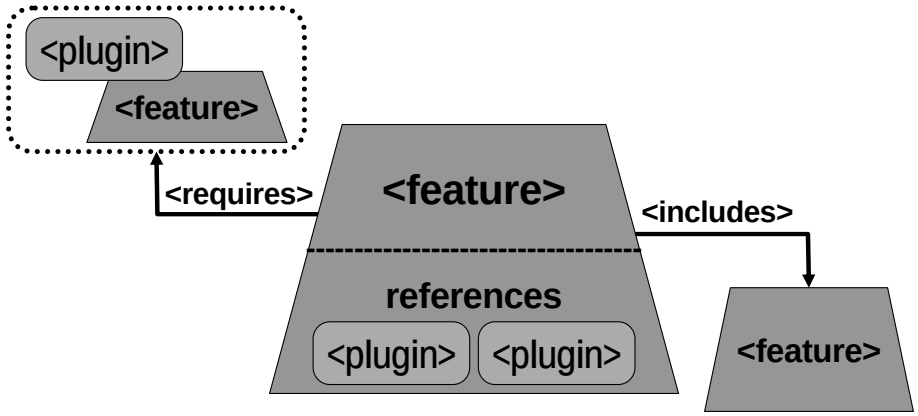
- For packaging plug-ins:
  - Organize plug-ins so that they can be installed and managed by the Update Manager
  - It is possible to brand features
- For installing plug-ins:
  - The user can choose to disable or enable features.
  - Features can be nested to manage the source of service for a set of features.
  - Features are installed and managed using the Update Manager.




5

IBM

# Feature Structure




- Required plug-ins and features used to validate feature install and enablement
- Referenced plug-ins are managed and serviced by referencing feature
- Included features are serviced by the parent-defined update URL

6

# What are Install Sites?

- Install sites are the basic building block in a ***configuration.***
- An install site is a location on the file system where the `features` and `plugins` directories can be found.
- An install site is a single location on the file system, but the same location could be included in multiple configurations and in multiple Eclipse-based product installations.
- Types of install sites:
  - **Platform base site**: This is where Eclipse itself is installed. It always exists.
  - **Extension site**: Distinguished from other sites through the .eclipseextension file.
  - **Update site**: Distinguished from other sites through the site.xml file.


Update Site

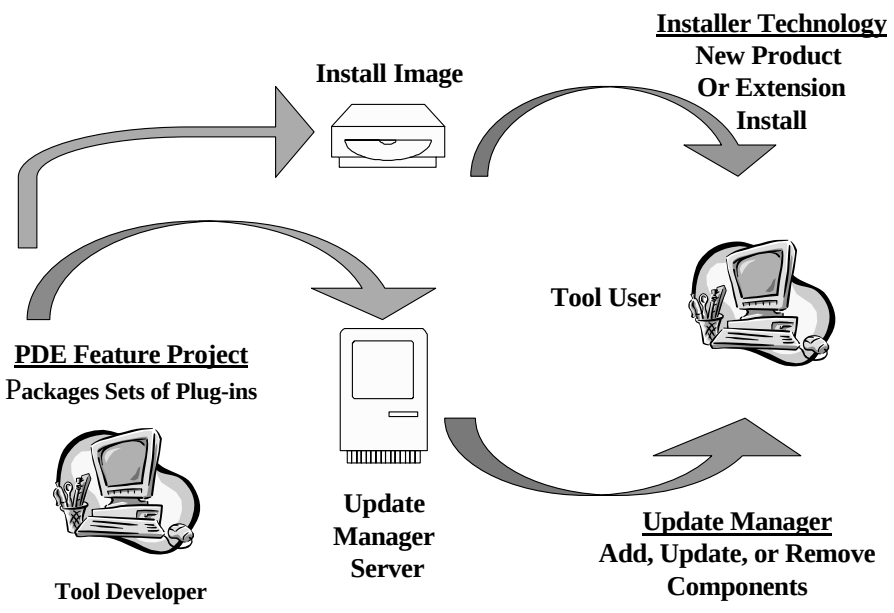
7

# What are Configurations?

- The Update Manager controls your Eclipse configurations.
  - Creates an initial configuration during startup if there is no existing configuration
  - Reads the active configuration
  - Manages changes that occur to the configuration
- A Configuration identifies:
  - What Install sites are accessible
  - What Features exist in each site
- Configuration information is saved in the `platform.xml` , which is filefound in the `configuration/org.eclipse.update` directory
- A Configuration applies to any workspace that might be accessed
- The default configuration is used when Eclipse is launched
  - `<eclipse_install>/eclipse/configuration`
- An alternate configuration can be specified using the `-configuration` startup parameter

8

# Feature Scenarios

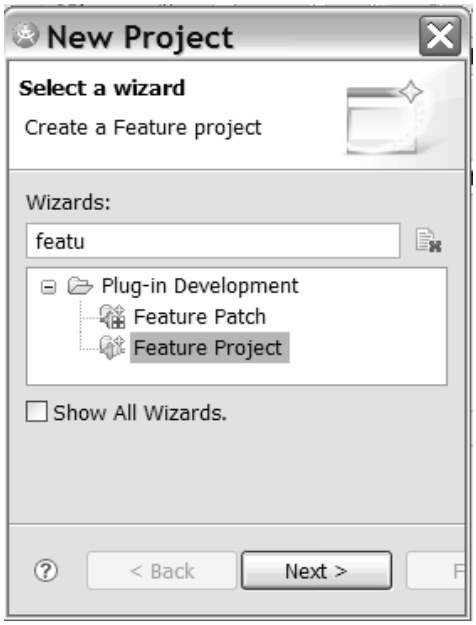



9

IBM

# Create a Feature

1. Open the New wizard dialog by hitting CTRL+N, filter for `feature`, and click **Next**.
2. Add project name and Feature Provider, and click **Next**.
3. Select required plugins and features, and **Finish**.
4. feature.xml opens in a multi-page editor similar to plugin.xml.




10

# Edit a Feature

- Overview page
  - Define Update URL
- Information page
  - Define Feature description
  - Define Copyright Notice
  - Define License Agreement
- Plug-ins page
  - Define included plug-ins
- Feature page
  - Define Features to be nested


11

# Package a Feature: Overview (1 of 3)

- Packaging plug-ins using features is required to install into Eclipse
- The process is driven by the build.properties, feature.xml, and plugin.xml files
- Features and their associated plug-ins can be packaged in two ways:
  1. Packaging for an Extension site
     - `eclipse` folder
       - `.eclipseextension` file
       - `features` folder
         - `<featureId_ver>` folder
           - `Feature.xml`
           - …
         - …
       - `plugins` folder
         - `<pluginsId_ver>` folder
           - `plugin.xml`
           - `<runtime>.jar`
           - …
         - …

12

# Package a Feature: Overview (2 of 3)

- Eclipse features and their associated plug-ins can be packaged in three ways:
  1. As a RAS asset
  2. As Packaging for an Extension site
  3. As Packaging for an Update site
     - `updateSite` folder
       - `site.xml` file
       - `Features` folder
         - `<featureId_ver>.jar`
         - …
       - `Plugins` folder
         - `<pluginId_ver>.jar`
         - ….

13

# Package a Feature: Overview (3 of 3)

- Packaging options include:
  1. Ant scripts using PDE
     1. Build for an Extension site
     2. Build for an Update site
  2. Export Features wizard
     1. Export for an Extension site
     2. Export for an Update site
  3. Site editor's Build All action
     1. Build for an Update site
  4. Ant scripts using AntRunner
     1. Build for an Extension site
     2. Build for an Update site

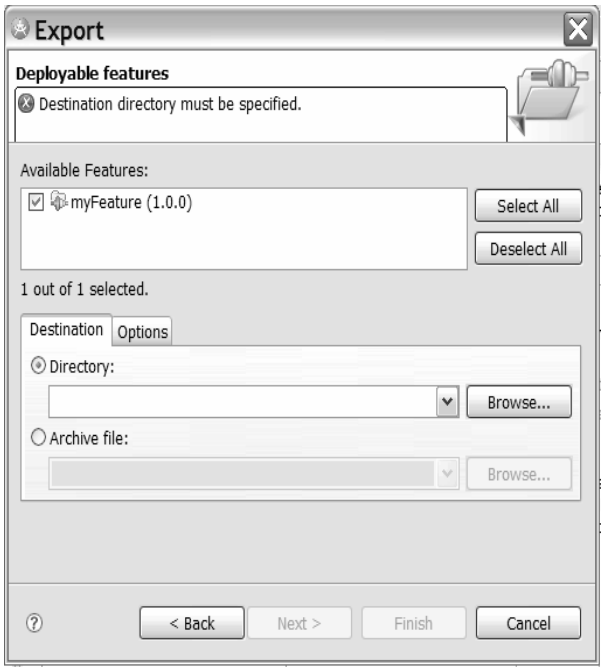
14

IBM

# Package a Feature: Export Features Wizard

The Export Features wizard provides three options:

1. Export as a directory structure
   - Builds for Extension site
2. Export as a single ZIP file
   - Builds for Extension site
3. Export as individual JAR archives
   - Builds for Update site




15

# Package an Install Site

- Two options exist for how Features can be installed in Eclipse. The result of these two techniques is the same.

1. Install Feature from an Extension site
   - Tool providers package their Features including an `.eclipseextension` file using **InstallShield**.
   - `.eclipseextension` content includes:

```
id=com.ibm.jdg2e.simplemodel.ui
name=JDG2E Simple Model UI
version=1.0.0
```

2. Install Feature from an Update site
   - Tool providers package their Features including a `site.xml` file to a **HTTP site**.
   - `site.xml` content includes:

```
<site>
   <feature url="features/com.ibm.jdg2e.simplemodel.ui_1.0.0.jar"
   id="com.ibm.jdg2e.simplemodel.ui" version="1.0.0">
      <category name="jdg2e.service"/>
   </feature>
   <category-def name="jdg2e.service" label="JDG2E Service">
      <description> JDG2E Service Description </description>
   </category-def>
</site>
```



# Summary

- Features are:
  - An installable unit of function
  - A packaging construct
- Features can:
  - Brand plugins
  - Nest other features
- Features can be:
  - Installed and managed using the Update Manager
  - Developed using PDE

17

## Where Are We?

- Eclipse Features
- **Reusable Asset Specification Archives**

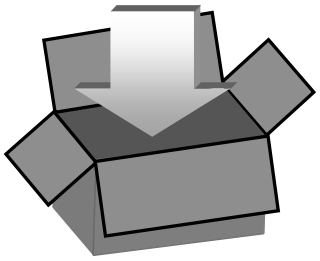

18

IBM

This section provides an overview of the Reusable Asset Specification.

## Reusable Asset Specification (RAS)

- The RAS provides a standard format for assembling, organizing, storing, and documenting reusable assets, including extensibility artifacts.
- Rational Software Architect supports the exchange of RAS assets:
  - RAS Import and Export
  - RAS documentation
  - .ras file format
  - Support for RAS repositories



19

IBM

Rational Software Architect uses the Reusable Asset Specification (RAS) to provide a standard way to package and extract a set of related files. A RAS asset is a RAS-compliant collection of related files or artifacts.

A RAS asset can contain many types of artifacts; for example, design and use-case models, pattern assets, Web links, code samples, text files, and test data. Assets targeted for long-term reuse benefit from good documentation that summarizes the asset's purpose, use, content, and context. Documentation plays a key role in helping the consumer determine if the asset satisfies his requirements.

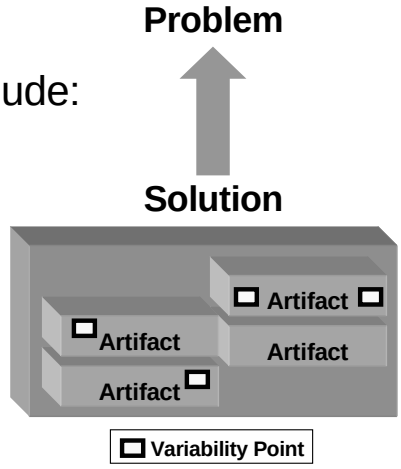
RAS assets provide the following benefits:

- A method to communicate software solutions easily.
- Organization of diverse, but related, files in a single package.
- Presentation of consistent information in all assets of the same type.
- Multiple and flexible keywords to search repositories for assets.
- Options to store and retrieve assets from one or more RAS asset repositories.
- Use of simple variations of the standard import and export functions to load and package assets.
- Maintenance of activities to ensure accurate file restoration upon import or export.

The RUP for Asset-Based Development plug-in describes the asset identification, asset production, and asset consumption process components of the asset-based development discipline. This plug-in is available for download from the “List of RUP Plug-ins” on IBM developerWorks: http://www-128.ibm.com/developerworks/rational/library/5823.html.

## RAS Assets

- A RAS asset contains:
  - Artifacts
  - Variability points
- Parts of a RAS asset file include:
  - RAS manifest
  - Asset profile
  - Activity task types




20

When you begin using pattern solutions in the development environment, you need to find a standard way to store and share them (along with other project artifacts). This mechanism for sharing artifacts is the reusable asset.

A **reusable asset** is an organized collection of artifacts that provides a solution to a problem for a given context. Assets clearly have much in common with patterns. Similar to a pattern, an asset:

- Includes instructions or usage rules, to minimize the time developers need to discover, analyze, consume, and test the asset.
- Includes standard documentation describing the development and business context in which the asset can be used.
- Can have **variability points**, like pattern parameters, that allow users to customize the asset for a specific project.

An asset is a more general concept than a pattern, since it is a collection of artifacts and not just a collection of model elements.

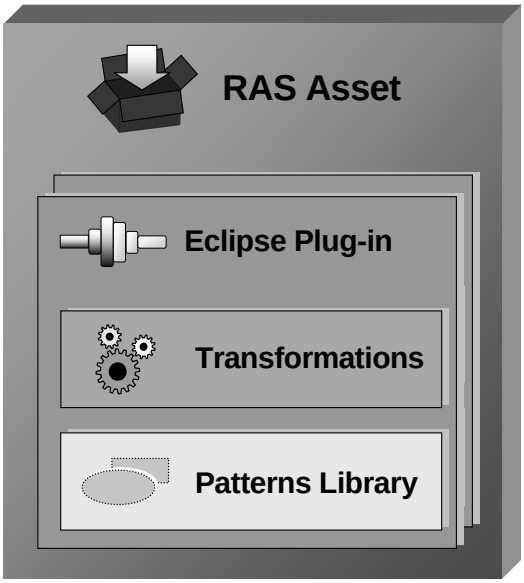
An asset can contain more than patterns. An asset for a development project might contain requirements, models, source code, and tests. Assets might also be used to package and share deployable components, Web services, frameworks, and templates.

The standard structure of reusable assets is the Reusable Asset Specification (RAS), an OMG standard. The Rational brand products use the RAS specification for exporting and importing assets to help with asset-based development. By default, a local repository is provided for storing and retrieving your files. A Pattern repository also comes preloaded in the Asset Explorer view. Additional repositories can be established using applications for Web-based access.

- **RAS asset manifest file:** The RAS asset is a compressed file that stores the files that make up the asset. At export, a manifest file is created (from the selected RAS profile file) and is included in every RAS asset's file.
- **Types of RAS asset profiles:** RAS asset profiles allow you to create different types of assets. A specialized profile extends the original contents of the default profile. Every RAS manifest must have a RAS profile associated with it.
- **Activity task types:** Activities should be modified only by users who are familiar with using the Reusable Asset Specification to hand code manifest files. Modifications to the RAS manifest files-generated activities can render them incompatible. Activities describe tasks the user should do to reuse the asset. It is recommended that you do not modify generated activities, but you are encouraged to add your own as needed.

## Packaging UML Patterns and Transformations

- Patterns with transformations can be grouped into pattern libraries
- Pattern libraries and transformations can reside in the same Eclipse plug-in
- Plug-ins can be grouped and exchanged in RAS format.




21

IBM

Patterns realize their maximum benefit from reuse and distribution as RAS assets. They can be exported as deployable plug-ins.

As patterns are created, the required meta (RAS manifest) files (which support the RAS packaging) are added to the pattern project, both to the individual patterns that it comprises and to the pattern library itself on export. The content of the manifest file is determined by a pattern's profile, which specifies the type of the meta file used to package (and also restore) RAS assets.

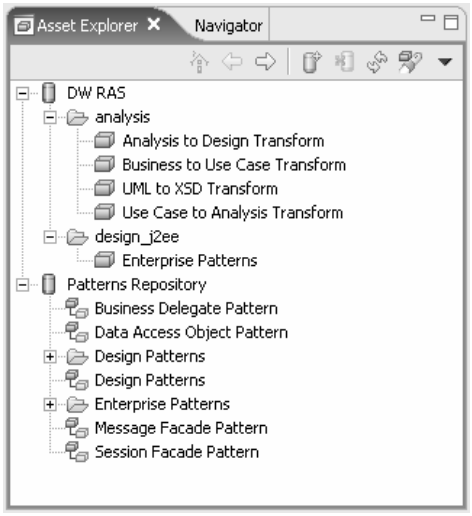
A pattern repository is always created for you whenever any pattern plug-ins are detected and available to your workspace. Patterns installed as plug-ins and patterns in other repositories all display in the pattern repository and the Pattern Explorer view.

Common RAS features, such as searching and adding groups (folders) are also available in the Pattern Explorer view. Thus, pattern functions can be accomplished without using the Asset Explorer view.

# RAS Repositories

## Repository Types

- DeveloperWorks
- Local
- Workgroup
- IBM® Rational® XDE™




22

IBM

Repositories provide a way to organize and manage assets so that they can be exchanged quickly and easily with developers and co-workers. You can easily add a repository to your repository list so that you can view, search, inspect, and import these assets. The RAS feature supports the following types of repositories; developerWorks, Local, Workgroup, IBM® Rational® XDE™, and Patterns.

- developerWorks repository: Contains new Rational Software Modeler Product assets, and is hosted by IBM on the developerWorks website.
- Local repository: Resides on your local personal computer, and does not contain any assets until you populate it.
- Workgroup repository: Can be any J2EE Web server repository. Note: Workgroup repositories run only on IBM® WebSphere® Application Server 5.1 or later.
- XDE repository: Contains IBM Rational® XDE assets. This format is provided so that you can use legacy assets developed and exported from Rational XDE.

Neil Boyette of IBM research has produced a RAS Repository for Workgroups for IBM® alphaWorks®, IBM's resource for emerging technologies. The Reusable Asset Specification Repository for Workgroups supports a variety of ways for users to retrieve information about the assets in the repository. These include searching and browsing with the RAS 1.0 standard interface, or with an enhanced interface (to be proposed for incorporation in the next version of RAS) that supports more complex queries. Administrators can publish assets to the repository, create and organize the logical view of the assets, and perform measurement tracking.

This repository is on IBM alphaWorks: http://www.alphaworks.ibm.com/tech/rasr4w

# Lab 15: Package Reusable Artifacts

- Given:
  - Project with Reusable assets
- Complete the following tasks:
  - Create a RAS Repository
  - Create RAS asset containing reusable assets
  - Test RAS asset

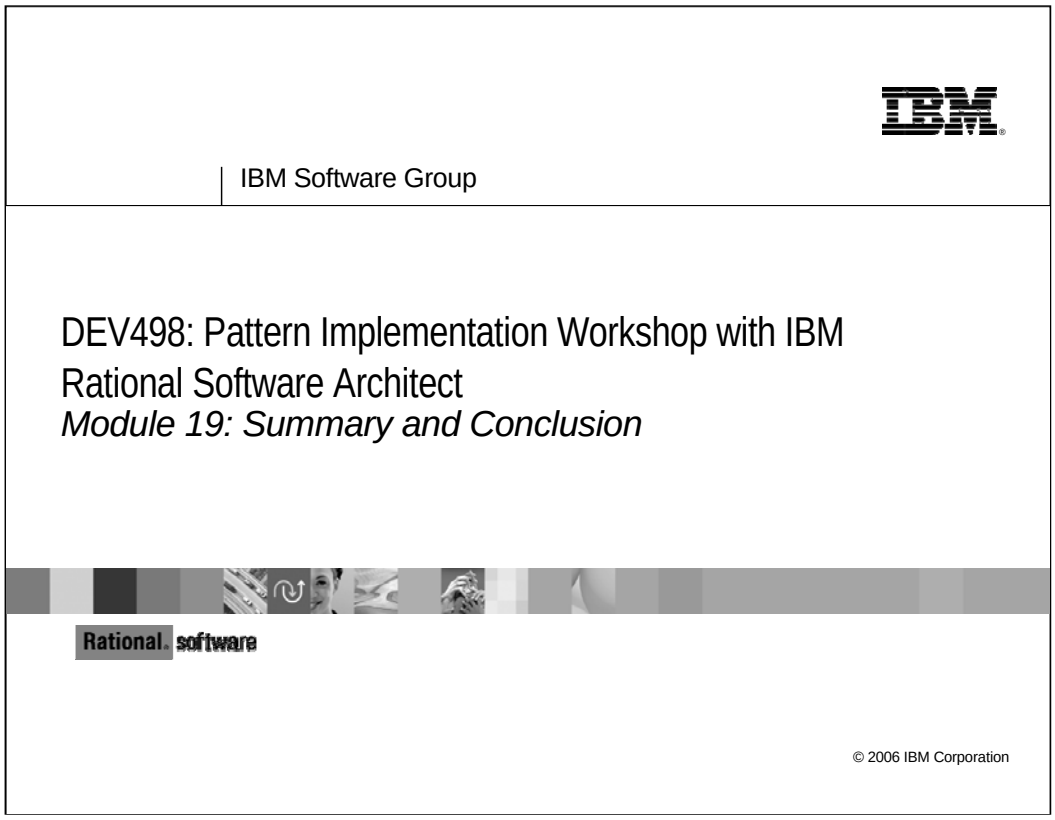
23

IBM

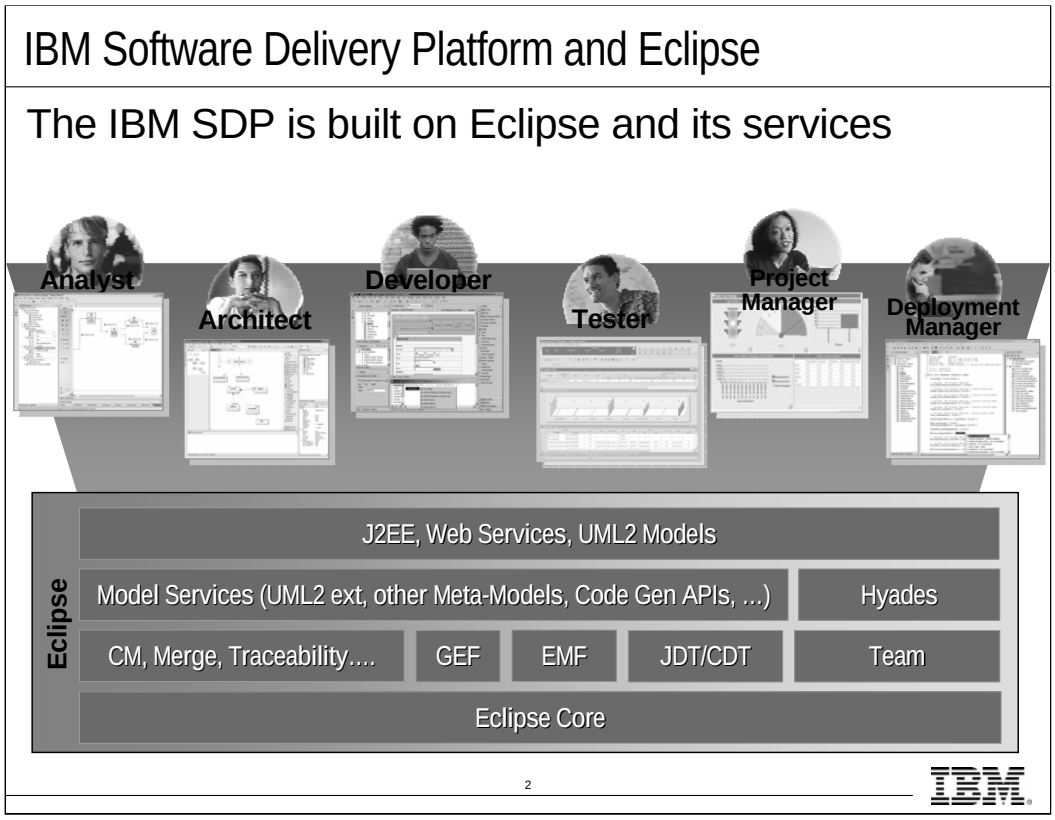
Complete Lab 15 in the student workbook.

## Review

- Why do we want to package the asset we build?
- Why wrap the plug-in with a feature?
- What is RAS?




## Contents




The Eclipse platform provides an open environment for enterprise development, with the capacity to support all phases of the application development life cycle, including analysis, requirements, design, development, testing, software configuration management, defect tracking, project management, and so on. The main components include a universal platform for development tool integration, and a Java development environment built with Eclipse. At the heart of the Eclipse platform is an extensive toolset with core capabilities, plus support for extensions through a plug-in architecture.

The components of a development tool chain based on the Eclipse framework are:

- **The Eclipse Modeling Framework:** A fundamental part of Eclipse, enabling the platform's modeling capabilities to interoperate with other tools and applications.
- **The Eclipse C and C++ Development Tools (CDT) project:** An open-source C and C++ development plug-in that leverages common open-source underlying tools such as gcc, gdb and make.
- **The Eclipse Test and Performance Tools Platform Project (TPTP)**: Provides a common user interface, standard data models, data collection and communications control, as well as remote execution environments. Can be extended for solution-specific tooling and runtimes.
- **The Graphical Editing Framework (GEF)**: Allows you to easily develop graphical representations for existing models.
- **Java Development Tools (JDT):** Java development tools, along with the Eclipse technology, create applications that run on real-time operating systems and embedded environments.

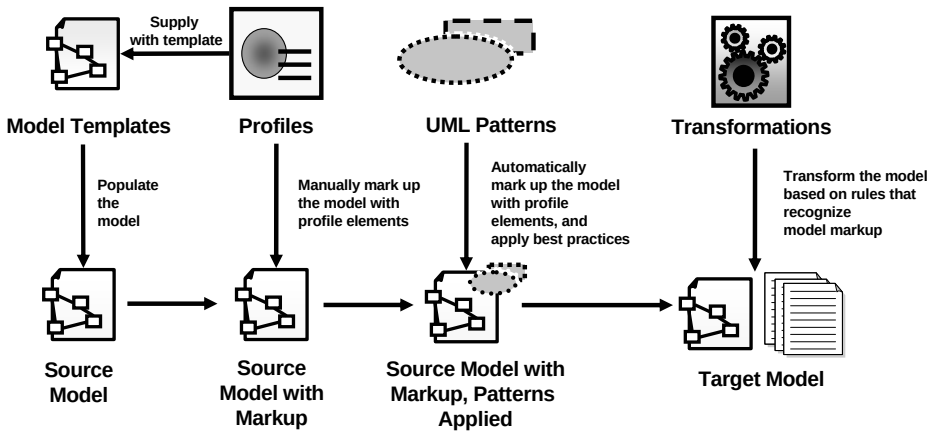
## Artifacts for Reusable Assets in Rational Software Architect

- Profiles
  - Provide custom stereotypes
  - Are often used in patterns
  - Are required for transformations
- Model templates
  - Allow you to create a new model based on a pre-existing structure
  - Can provide a model structure consistent with related patterns and transformations
  - Can be distributed with a custom profile applied
- UML Patterns
  - Are developed as Eclipse plug-ins
  - Add or change structures in the model
  - Are available in libraries for different types of development projects
- Transformations
  - Are developed as Eclipse plug-ins
  - Transform model elements based on a transformation definition
  - Are applied to specific elements or whole models
  - Work as part of pattern solutions

3

## Integration of Artifacts in Reusable Assets

Profiles, model templates, UML patterns, and transformations can be used together to increase the ROI of your extensibility investment.

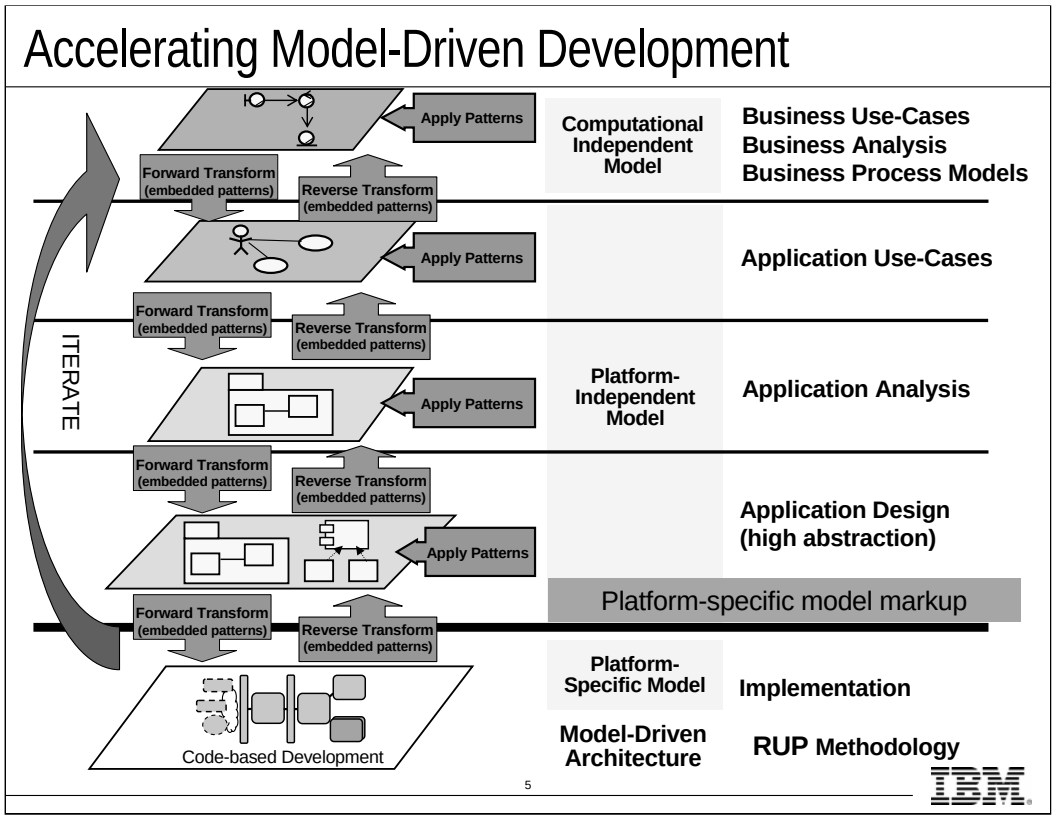



4

IBM

As you work with Rational Software Architect in your environment, you will come across situations where plain UML is not able to model the elements of your domain sufficiently. In addition, there will be patterns of usage that will accompany these domain-specific elements. UML profiles can be developed in Rational Software Modeler or Rational Software Architect for these situations.

Creating a UML pattern that can understand and use the domain-specific elements of your profile will help in ensuring that users are following best practices for your organization. As a final step in this workflow, the user would send the model through a transformation. Ideally, the model elements would then be updated according to the profile, with elements structured in a way that makes the best use of those model elements. The transformation will understand the domain-specific elements, and will produce an output model that reflects this understanding.




A code-generator is an important component of Model Driven Development (MDD). The goal of MDD is to describe a software system using abstract models (such as EMF/ECORE models or UML models), and then refine and transform these models into code. Although it is possible to create abstract models, and manually transform them into code, the real power of MDD comes from automating this process. Such transformations accelerate the MDD process. The transformations can capture "best practices" and can ensure that a project consistently employs these practices.

However, transformations are not always perfect. Best practices are often dependent on context - what is optimal in one context may be suboptimal in another.

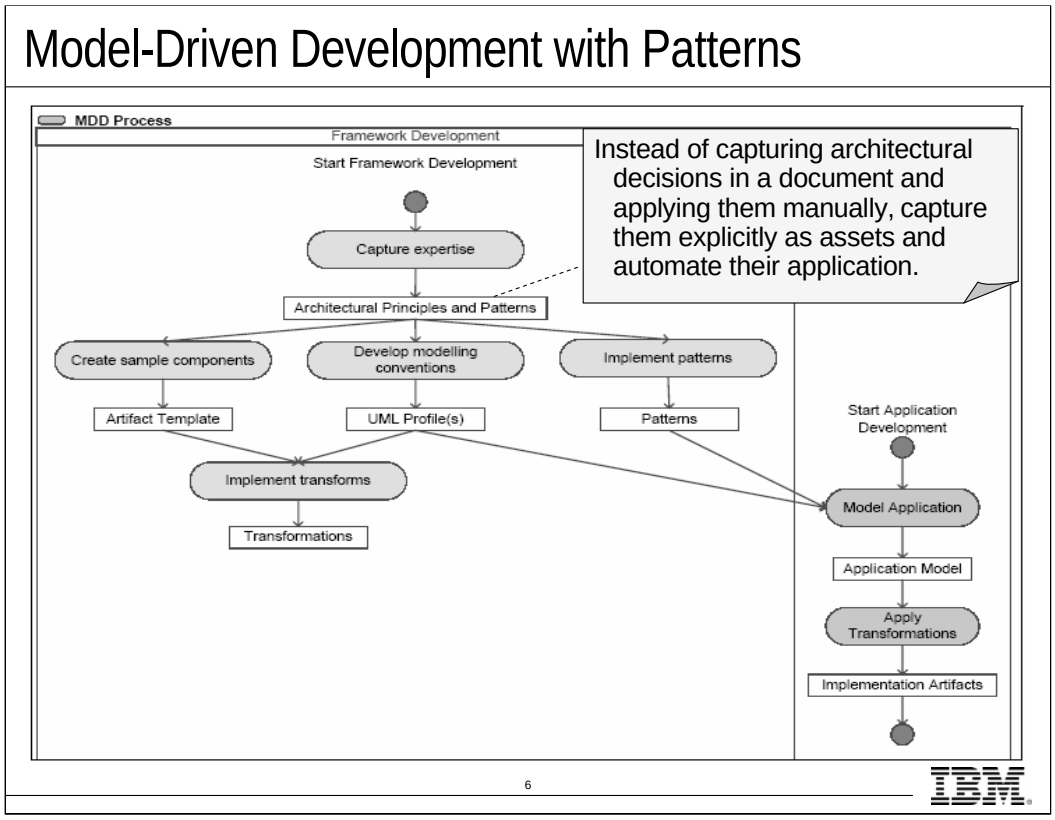
Rational Software Architect is designed to support MDD, the development of the appropriate models to facilitate all development activities and stages in the lifecycle, plus tools to transform models to move development work forward.

An analyst might begin by modeling the business domain in Rational Software Architect to define the key products, deliverables, or events. The analyst can then create a use-case model to define the actors, system boundary, and use cases the system will support.

The architect then uses Rational Software Architect to create a platform-independent design model from the use-case model. This model or set of models can be transformed in platform-dependent implementation models (including code and UML) with the assistance of visual development tools, such as:

- UML editors for Java, C++, or data
- Site Designer
- Page Designers

As each new stage of development begins, transformations can create more detailed models that are incrementally closer to the target platform and infrastructure. Transformations can be designed to include traceability so that you can query the target model, using elements from the source model to find elements in the target model. This feature is currently built into the UML to Java transformation that comes with Rational Software Architect. After the transformation is complete, you can right-click a model element in the design model and perform a query to find the associated Java code.



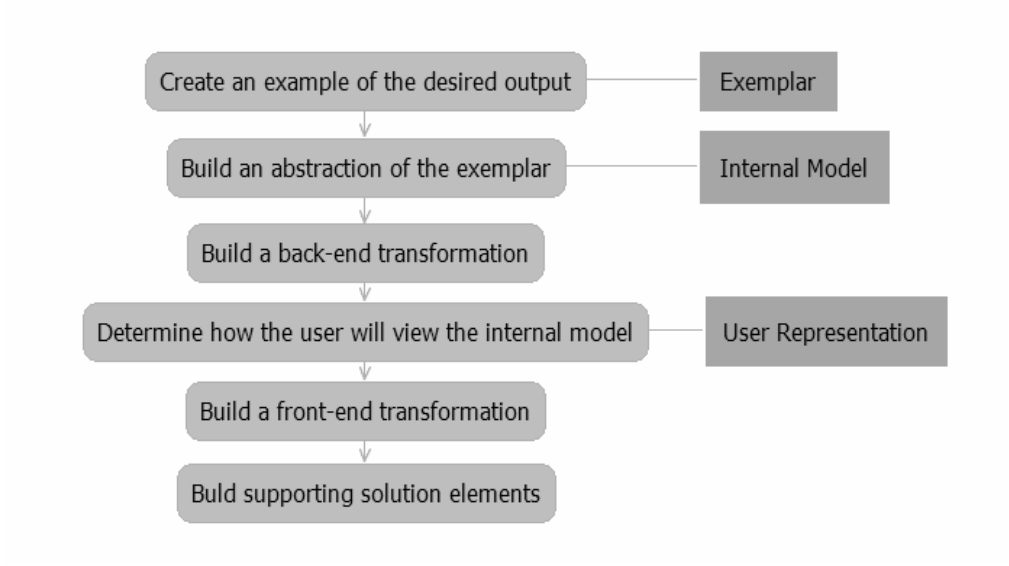

There the two distinct types of activity in the MDD process:

- **Expertise Capture and Automation:** This is where you build the MDD framework that partially automates the development of software that follows a particular architectural style.
- **Application Development:** This is where you apply your chosen MDD framework to build software components, applications, and solutions. These activities are typically performed by different groups of people and require different skills. Rational Software Architect supports both sets of activities. You use Rational Software Architect to build UML profiles, patterns, and transformations that are then used to customize Rational Software Architect to provide an MDD framework.

There is no magic to MDD. Someone must come up with a set of modeling conventions that are suitable for the software under development. Someone must also develop transformations that can automate the generation of code from models that follow these conventions. The key dependencies between the two streams of activity are as follows:

- UML profiles and patterns must be available when application modeling begins. In some cases, this dependency is managed in an iterative manner, with profiles and patterns that address some aspects of design being made available before others.
- Transformations must be available in order to generate implementation artifacts. In some projects, the target platform and the transformations are selected at the start of the project. In others, this decision is deferred.

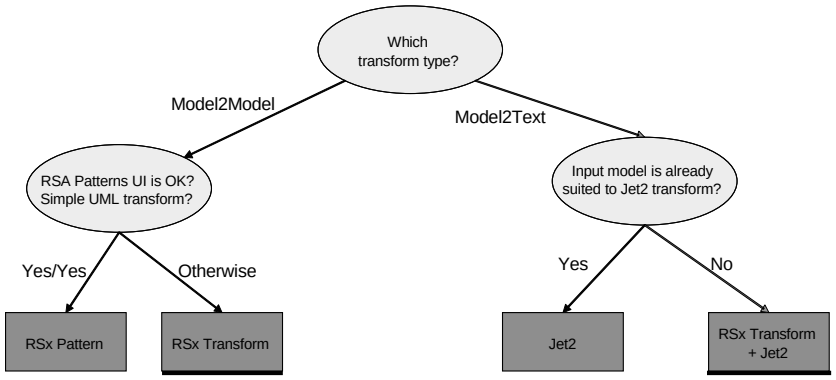
# The Asset Development Process




7

IBM

# Choosing the Kind of Pattern Implementation



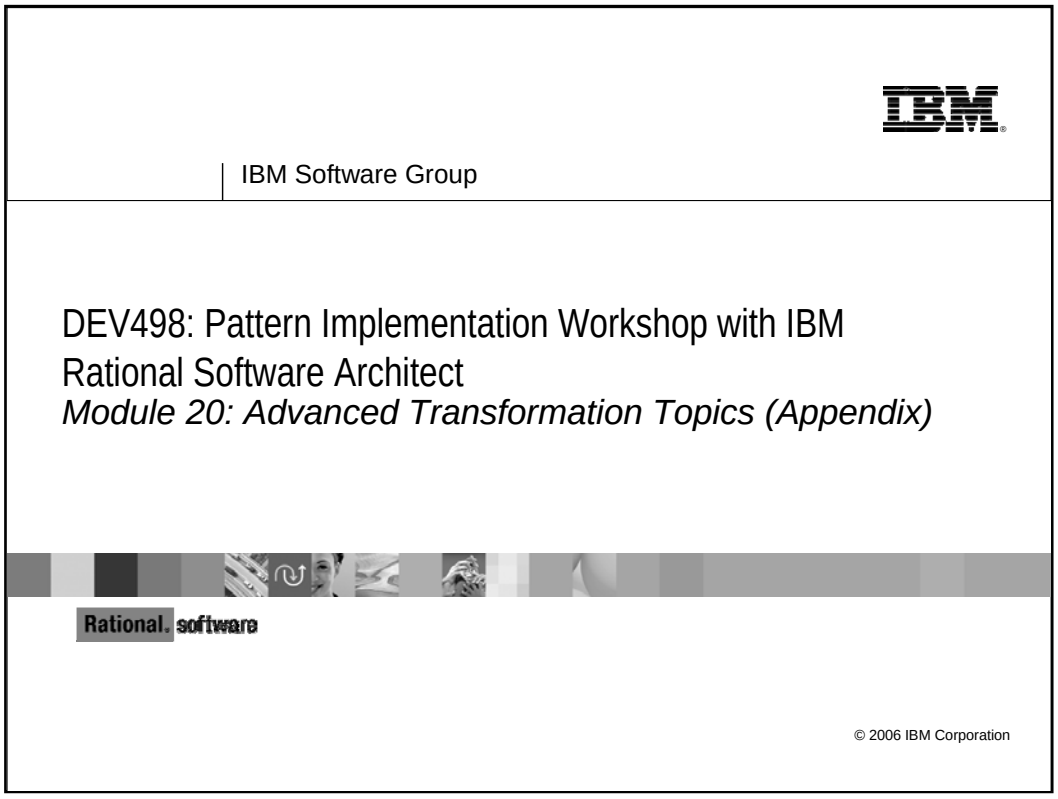

8

## Class Discussion

- How does the use of reusable assets help in the design of a software solution?
- How do you plan to apply reusable assets in your current projects?
- How would you evolve the artifacts from the workshop?

9

IBM




## Contents

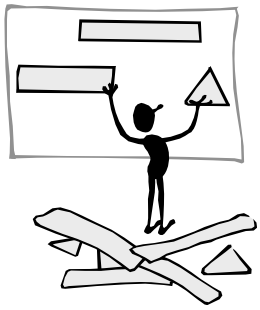
# Advanced Transformation Topics

- Objectives:
  - Describe how to:
    - Extend transformations
    - Clone transformations
    - Customize the Transform GUI
    - Decide when reverse transformations are needed



# What Else Can Be Done to Transformations?

- Create Transformation extensions
- Clone a Transformation
- Call a transformation from a menu or other plug-in
- Enhance the Transformation UI
- Include a reverse transformation



3

## Where Are We?

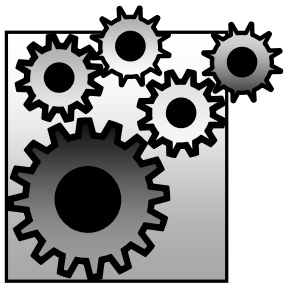
- **Transformation Extensions**
- Cloning Transformations
- Enhance the Transformation UI
- Reverse Transformations


4

IBM

# Transformation Extensions

- Transformations in Rational Software Architect, as Eclipse plug-ins, are designed to be extended
- You can modify a transformation to add customized behavior
  - Generate additional items from your model according to your own code standards
  - A better alternative than creating your own transformation from scratch



5

IBM

# Types of Extension Points

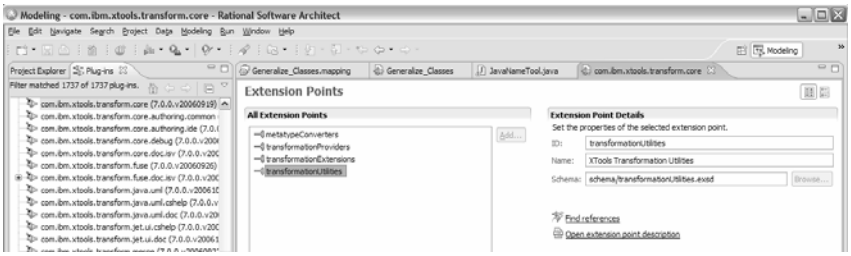
- Metatype Converters
  - This extension defines a metatype converter, which allows new metatypes to be defined and used by transformations and their properties.
- Transformation Providers
  - This extension point facilitates the configuration of providers for the transformation service. The transformation service enables Xtools clients to register model transformations. These transformations can be used to convert the data from one model into a different model.
- Transformation Extensions
  - This extension point facilitates the configuration of extensions to transformations that are defined by transformation providers registered with the transformation service. Using a transformation extension, a client can extend the behavior of an existing transformation.
- Transformation Utilities
  - This extension point lets users define and register transform utilities that can be used by transformations.

6

# Extending Transformations

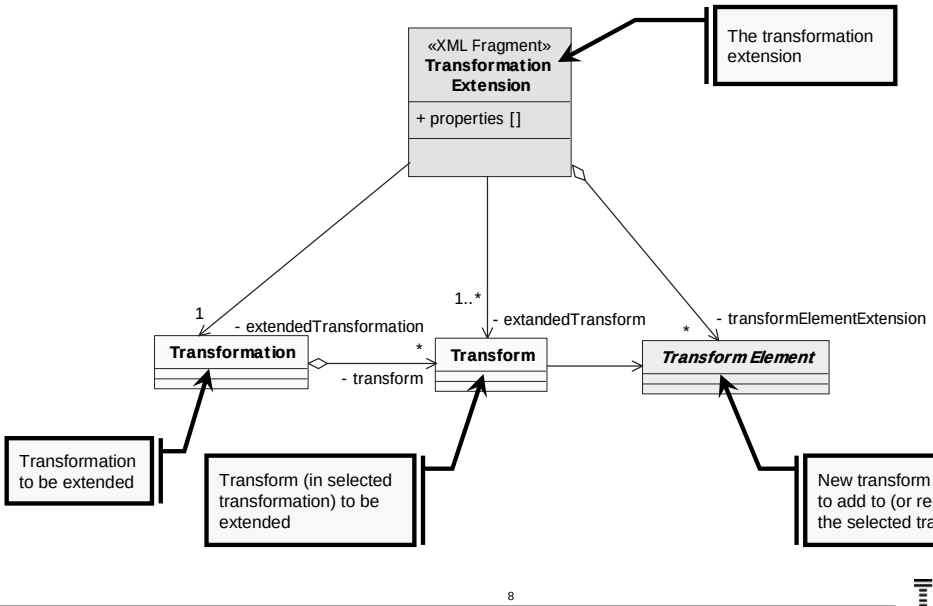
Extension point:
**com.ibm.xtools.transform.core.transformationExtensions**

- Used to:
  - Define new properties
  - Define new transforms
  - Define new rules
  - Define new extractors
  - Add new transforms, rules, and extractors to an existing transformation
- Does not create a new transformation, but adds behavior to the existing one
- When you run the transformation, all of its extensions are run
  - You must manually disable any plug-ins that you do not want to run




7

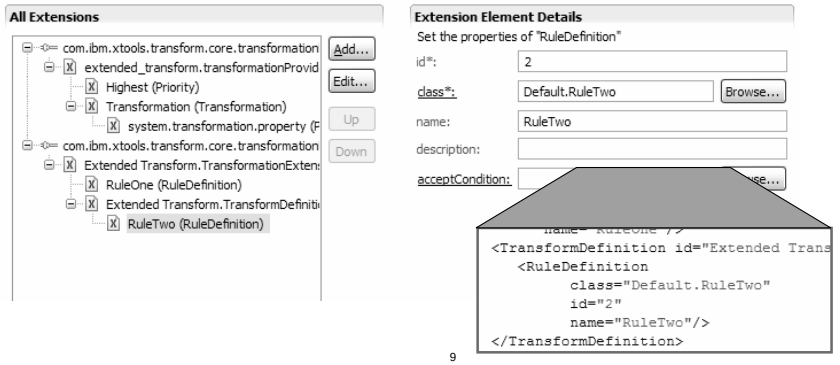
# Transformation Extension




8

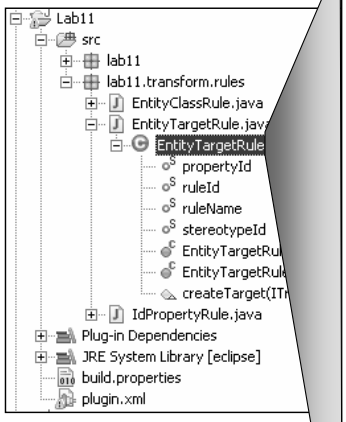
# AddRule Element

- AddRule element in the plug-in manifest specifies a rule and where it should be inserted
- AddRule has the following attributes:
  - **Index:** Shows where the rule is inserted among existing rules
  - **id:** Determines which rule is added. Must match the id of an existing RuleDefinition




9

# Extending a Transformation Rule




```
public EntityTargetRule() {
    super();
    setId(ruleId);
    setName(ruleName);
    setAcceptCondition(new HasStereotype(stereotypeId));
}

/**
 * @param id
 * @param name
 */
public EntityTargetRule(String id, String name) {
    super(id, name);
    setAcceptCondition(new HasStereotype(stereotypeId));
}

/* (non-Javadoc)
 * @see com.ibm.xtools.transform.core.AbstractRule#createTarget(com.ibm.xtools
 */
protected Object createTarget(ITransformContext context) throws Exception {
    NamedElement source = (NamedElement)context.getSource();
    Stereotype stereotype = source.getAppliedStereotype(stereotypeId);
    Object propertyValue = source.getValue(stereotype, propertyId);

    //setup default file name
    String newTargetFile = "Entity" + source.getName() + ".txt";
    if (propertyValue != null){
        newTargetFile = (String)propertyValue + ".txt";
    }

    //overwrite property targetFileName
    context.setPropertyValue("targetFile", newTargetFile);

    return null;
}
```

10

# Adding Transformation Extensions

- To discover how and where to extend a transformation, you need to rely on transformation documentation
  - Easy to extend your own transformations or transformations you have source code available
  - Hard to extend 3rd party transformations if no documentation is available
- Transformations designed to be extended must have well-documented extension points.

# Issues with Transformation Extensions

- Transformation extensions extend **all** instances of the transformation in the workbench
  - Can make customizable:
    - Using transformation properties
    - Using profiles
- Multiple extensions can extend the same transformation
  - Difficult to predict how all extensions will interact within the transformation
- Sometimes difficult to know where to insert new rules, and so on
  - It is necessary to know the ID of the transformation to extend
  - It is necessary to know the number of the rules of a transformation to determine the index to extend the transformation
- Extensions should have very specific accept conditions to prevent unintended side-effects on existing transforms

12

## Where Are We?

- Transformation Extensions
- **Cloning Transformations**
- Enhance the Transformation UI
- Reverse Transformations


13

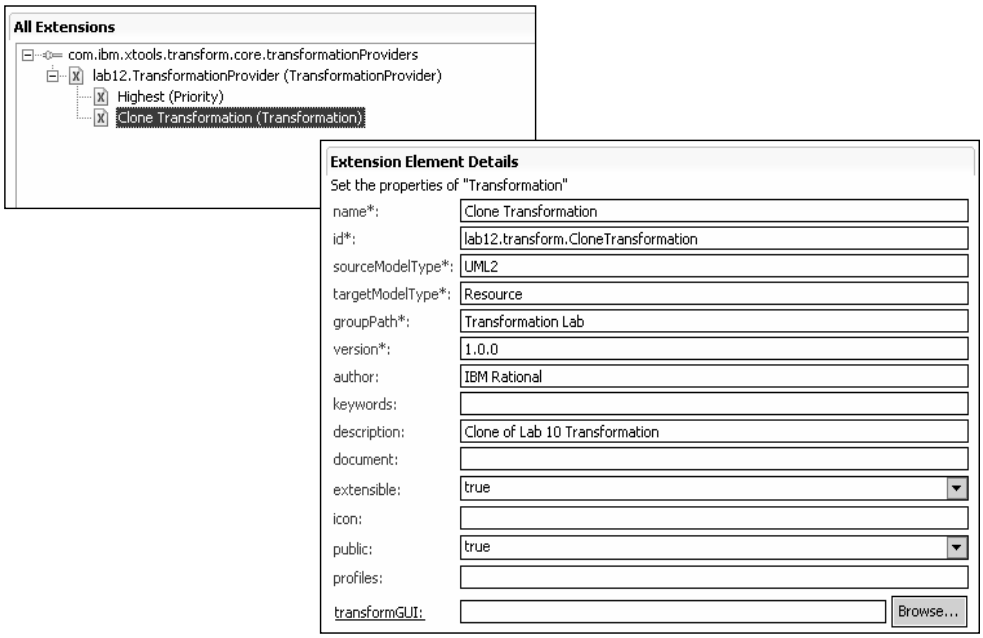
# Cloning Transformations

- Instead of simply extending an existing transformation
  - Copy a transformation and extend it to leave the original transformation available without the extension
    - Requires new TransformationProvider
    - New Transform class
      - Get original transformation from Transformation Service
      - Add original transformation to new Transform
    - Extend the new transform
- Works better if placeholders for extensions are defined for the original transformation

**Good Practice**

14

# Cloning Transformations




15

# Cloning Transformations Example

```
public class CloneTransformation extends RootTransform {


    /**
     * @param info
     */
    public CloneTransformation(ITransformationDescriptor info) {
        super(info);
        Transform transform = getClone();
        initialize(transform, false);
    }

    /**
     * Build this transform from the Lab 10 transform
     *
     */
    protected Transform getClone(){
        ITransformationDescriptor descriptor =
            TransformationServiceUtil.getTransformationDescriptor("lab10.transform.ClassToTextFile");
        Transform clone = (Transform)TransformationServiceUtil.createTransformation(descriptor);

        //add EntityClassRule
        Transform classTransform = (Transform)clone.findTransform("lab10.transform.OutputClassTransform");
        classTransform.add(1, new EntityTargetRule());
        classTransform.add(2, new EntityClassRule());

        //add IdPropertyRule
        Transform propertyTransform = (Transform)clone.findTransform("lab10.transform.OutputPropertyTransform");
        propertyTransform.add(0, new IdPropertyRule());

        //add clone into this
        //add(clone);
        return clone;
```

16

# Where Are We?

- Transformation Extensions
- Cloning Transformations
- **Enhance the Transformation UI**
- Reverse Transformations


17

# Enabling Custom Transformation UI

- Before a transformation author can customize the UI for a transformation, he must first inform the Transformation Service. This is easily accomplished by doing the following:
  - Create a class that is derived from AbstractTransformGUI and override the appropriate methods, such as `getConfigurationTabs()`
  - In the transformation descriptor in XML, add the **transformGUI** attribute where the value is the fully qualified class created above

18

# Adding New Configuration Tabs

- The method getConfigurationTabs() returns an array of configuration tabs to be displayed when a configuration for the associated transformation is selected. This list should include the three default tabs, where the **Target** tab usually comes first and the **Common** tab comes last.
- Each custom tab should be derived from AbstractTransformConfigTab and should be in the middle of the configuration tab list returned by getConfigurationTabs(). There are two key methods of this class that must be implemented:
  - populateContext(ITransformContext) saves the data from the tab's UI controls by defining one or more properties in the context with the appropriate values. These property values should be defined in the manner expected by the transformation when it executes.
  - populateTab(ITransformContext) resets the data for the tab's UI controls by examining one or more properties defined in the context.

19

# Filtering Displayed Source and Target Objects

- Although the source and target model types defined in the transformation descriptor enable the UI to filter the available source and target objects for the transformation, you may wish to provide additional pruning of the selection tree.
- There are two methods in AbstractTransformGUI for deciding if an object is to be displayed in the corresponding selection tree:
  - showInSourceTree() is called before an object is added to the transformation's source selection tree. The method enables the tree to be pruned. True should be returned if the given object is valid, or if it might contain a valid object. The method should return false if the object and all of its contained objects are invalid.
  - showInTargetContainerTree() is called before an object is added to the transformation's target selection tree. This method prunes the object in the tree in the same manner as the source tree.

20

IBM

# Where Are We?

- Transformation Extensions
- Cloning Transformations
- Enhance the Transformation UI
- **Reverse Transformations**

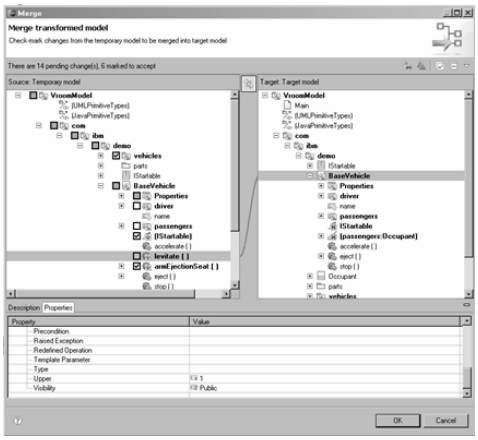

21

# Model-code Reconciliation and Reverse Transformations

## Benefits

*Allow globally distributed teams to work on design and implementation, and to reconcile results*

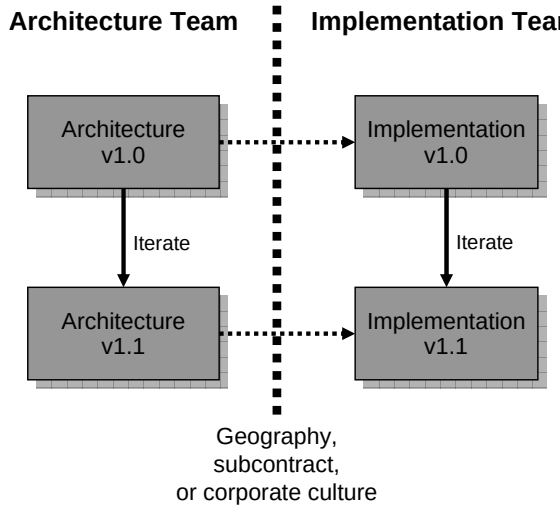
- **Enhanced difference and merge capabilities in Version 7.0**
  - Reverse transform code to model
  - Reconcile models
  - Merge resulting model
  - Forward transform model to code
- **Reverse engineering for code-to-model transformation**
  - Reverse transformations for Java, C++




22

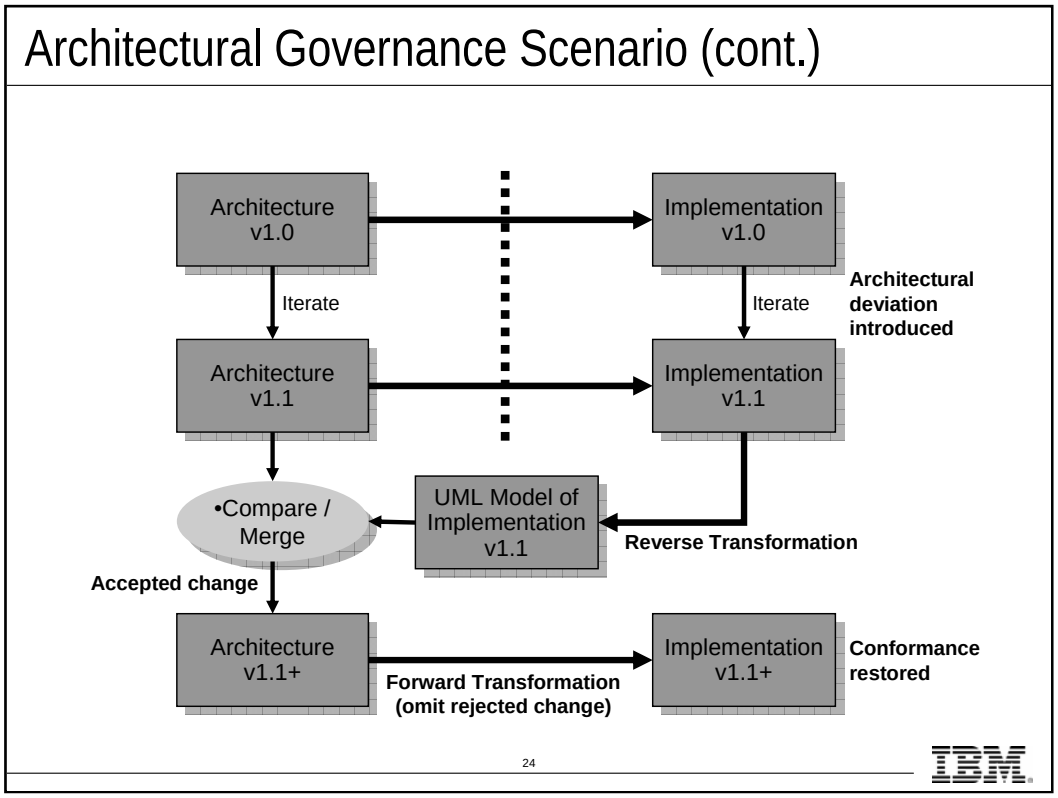
While there have been capabilities to harvest existing code into UML models with IBM Rational tools in the past , there has been no comprehensive way to differentiate and merge models and code. New transformations in V7 and later include the ability to reverse engineer code to UML models, reconcile differences, and merge the models together before forward engineering the merged architecture back to code. This allows globally distributed teams to work on design and implementation, while being able to ensure architectural integrity.

# Architectural Governance Scenario




- Will implementation conform to specification?
- How do you account for both changing over time?

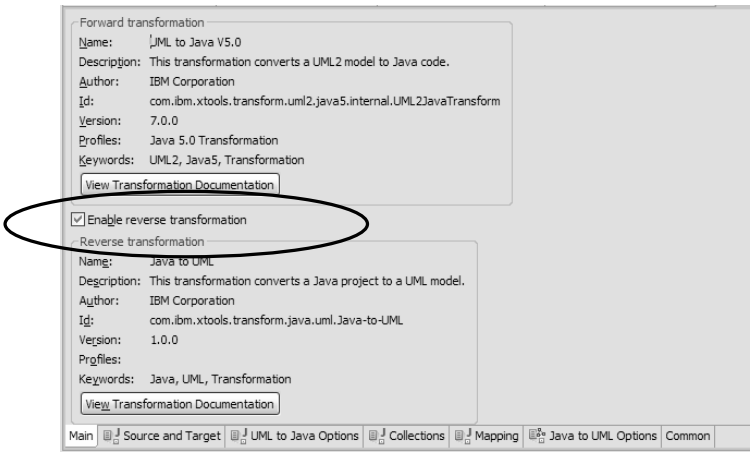
23




Some architectural deviations introduced by the implementation team may be accepted as improvements. Others may be rejected due to “bigger picture” concerns.

# Reverse Transformation Configuration

- When configuring a UML-to-Java or a UML-to-C++ transformation, you can choose to enable the corresponding reverse transformation.



# Running the Reverse Transform

- A reverse transformation allows the developers and designers to make changes to either the code or the model, and to keep those changes in sync.
- Running from code to a model could add implementation details to your model.
  - After the transformation runs, a dialogue will allow you to select the changes to apply.

## Further Information

- Rational Software Architect Help Topics
- Web resources
- Literature


27

IBM

**Rational Software Architect Help Topics**

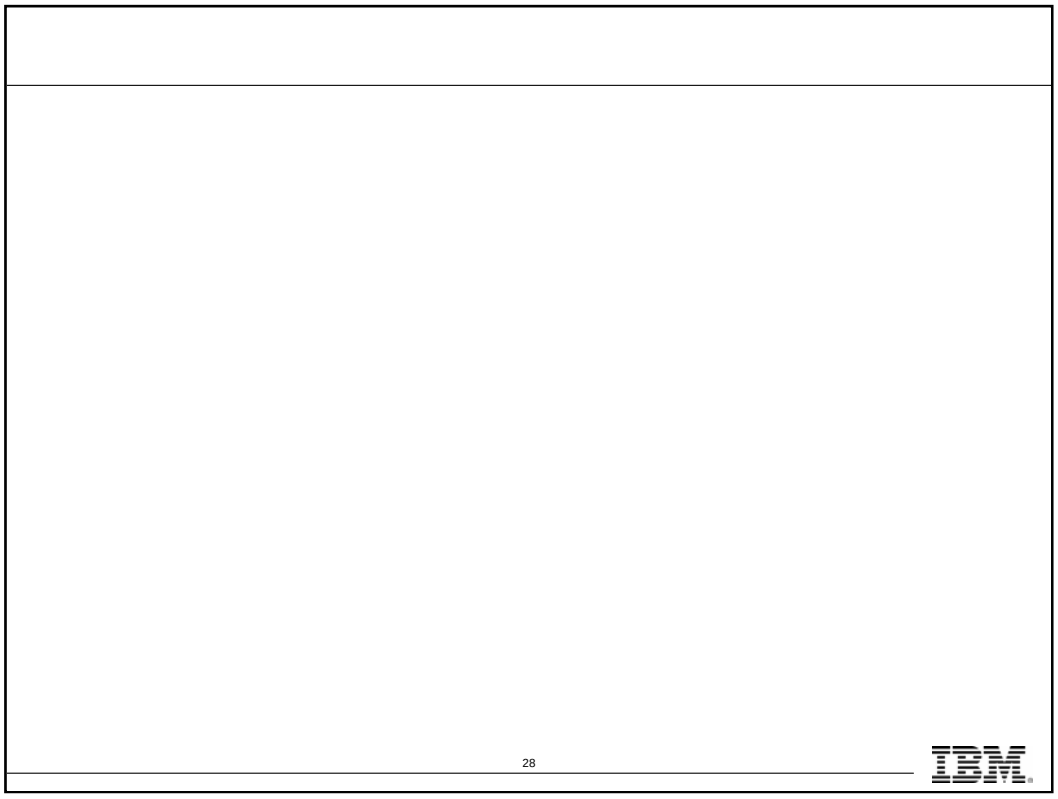
- Extending Rational Software Architect Functionality

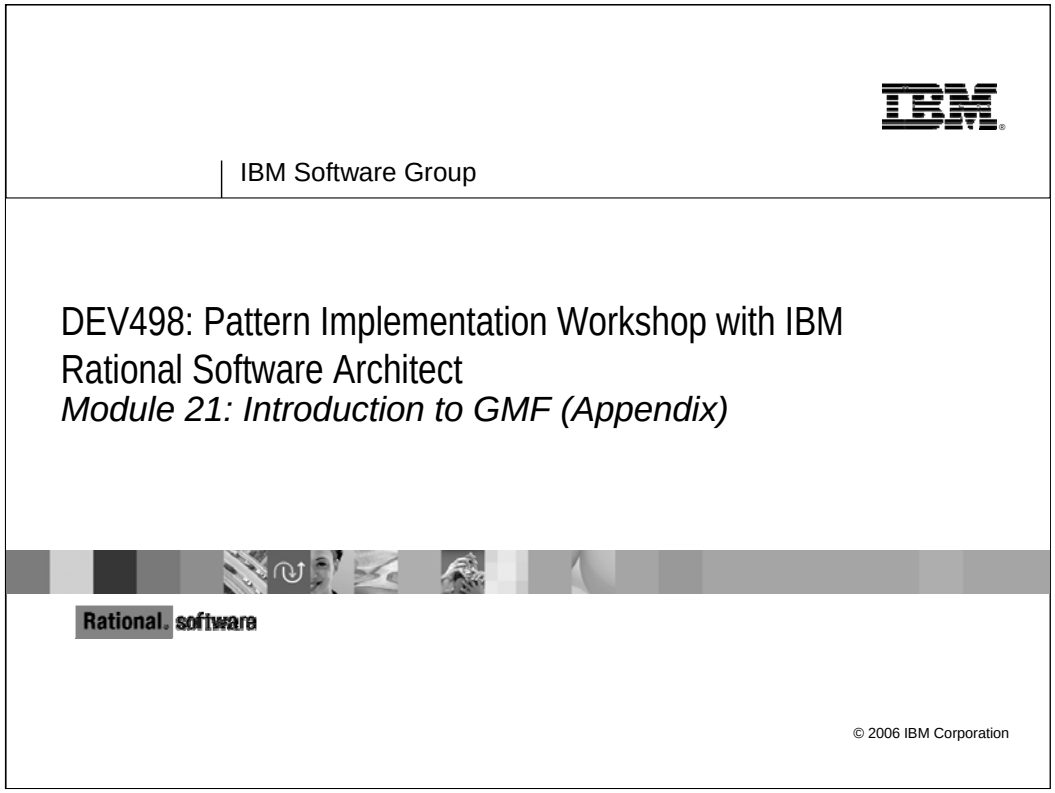
**Web Resources**

- "Extending the UML to Java Transformation in Rational Software Architect." http://www-128.ibm.com/developerworks/rational/library/05/802_uml/

**Literature**

- Frankel, David S. *Model Driven Architecture: Applying MDA to Enterprise Computing.* Indianapolis: Wiley Publishing, 2003.






## Contents

# Introduction to GMF

- Objectives:
  - Describe GMF
  - Understand how you can use GMF along with JET2
  - Understand DSL

2

## What is GMF?

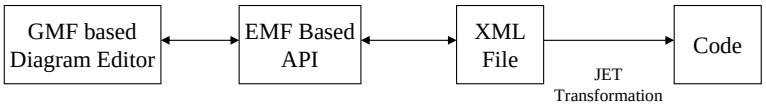
- GMF = Eclipse Graphical Modeling Framework
- Ability to create totally customized Diagram Editors
- May use UML or EMF (Eclipse Modeling Framework) and XML-based data
- Resulting diagrams have very similar look and feel to native Rational Software Architect diagrams

3

Remember that it is very easy to wrap XML data into an EMF-based API, and then create a GMF-based Diagram Editor for it.

## GMF Can Enhance JET

- Remember that JET transformations take XML (or EMF) files as input
- You can use GMF to create custom Graphical Diagram editors for JET input files

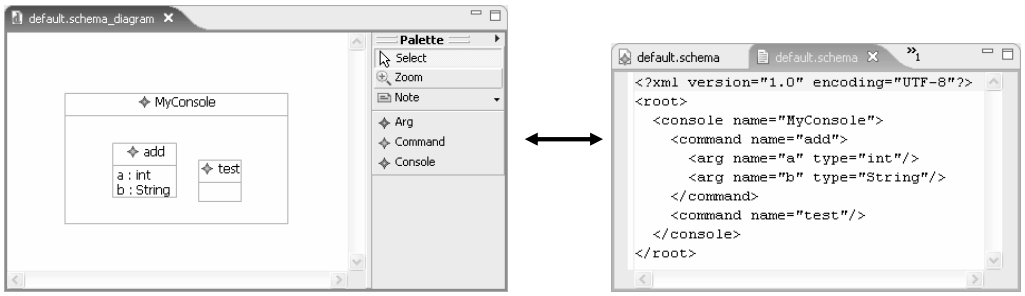



4

GMF can be used for many different scenarios. One possible scenario is to create a custom graphical Diagram editor for JET input files. GMF can be used to create a graphical editor for any XML file.

# Example

- Custom Diagram Editor to edit Console Transformation's input XML files




GMF XML

5

The diagram on the right is a custom Console Transformation Input editor. The diagram on the right is the resulting XML file.

## Overview of How GMF Works

- Use a set of GMF Wizards and Editors to define and generate a new Diagram
- Then write Java code to extend and refine the generated source code as needed
- The illustrated example is only using generated code
  - Took about one hour to create

6

# Lab 1: Run Example

- Given:
  - A Project Interchange file that contains a pre-built GMF based editor
- After completing this lab, you will be able to:
  - Use a GMF-based editor
  - See how GMF-based editors can edit XML files that can be used with transformations

7

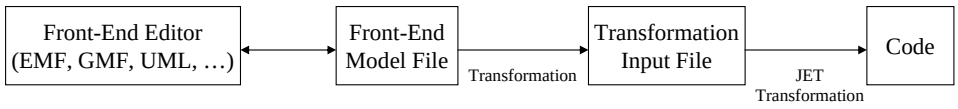
## Two-Phase Transformation

- Recommendation: When designing a transformation, derive its input model from the transformation
  - Better not to use existing model as input to transformation
- Recommendation: When designing a front-end model for users, derive the model from the user's perspective for ease of entry and maintenance
  - The model may be UML, XML, EMF, GMF, and so on
  - The model may already exist
- Result: Front-end model may be different from the transformation's input model, which is OK

8

# Two-Phase Transformation (cont.)

- Use another transformation to transform the front-end information into the Transformation's input (back-end model)
  - The front-end transformation may use JET
  - The front-end transformation may use Rational Software Architect's model to model the transformation engine
  - Generally, you should design the front-end transformation to automatically run the back-end transformation




9

# Introduction to DSL

- Domain Specific Language
  - A custom programming language or graphical modeling language designed to support a (domain) specific task
- In contrast to
  - Generic languages like Java and C++
  - Generic modeling languages like UML
- Examples
  - The sample Console input model is an EMF-based DSL for building Console Applications
  - The sample Console GMF Editor is a graphical DSL modeling language for building Console Applications

10

## How to Implement DSL with Rational Software Architect

- UML with UML Profiles
  - UML can be extended and customized using Profiles
    - Profiles add Stereotypes, additional model data, and additional model validation
  - Lets you extend and customize UML to create a DSL
- EMF with EMF and GMF based Editors
  - Using EMF, you can create a completely custom (XML-Based) language
  - Use EMF and GMF to create non-graphical and graphical editors for the language

11

## Mix and Match Implementation of DSL

- With Rational Software Architect, you can mix and match UML, EMF, and GMF in creating a DSL
- Examples:
  - Create a custom diagram for UML using GMF
  - Include EMF-based data inside of a UML model (emx file)

12

Rational Software Architect has very flexible support to use EMF, GMF, and UML together in various configurations. The examples listed are far from exhaustive.

## EMF/GMF versus UML/UML Profile-Based DSLs

- UML-based DSLs
  - Much easier to create
  - Much less flexible
- EMF/GMF-based DSLs
  - Much more flexible
  - Much harder to create and maintain

13

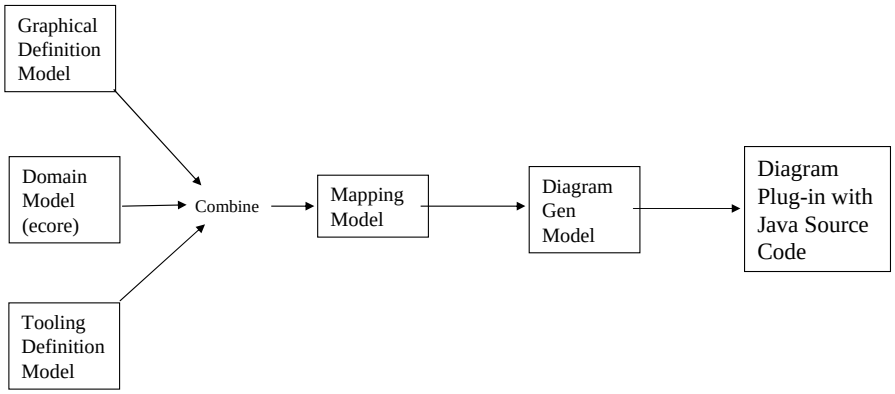
Note that UML has a lot of flexibility, but EMF/GMF has more.

## Optional: Technical details

- The remaining slides and lab describe GMF in more technical detail



## Overview




Note that in Rational Software Architect V7 and later, all of the Models have model type-specific non-graphical model editors.

15

The different models are explained in the next slides

## Domain Model (ecore)

- Defines the data model for the custom editor
- File extension is ecore
- This is any EMF data model which is an input into the GMF process

16

# Graphical Definition Model (gmfgraph)

- Defines the graphical elements for the custom editor
  - What are the nodes, compartments, connectors, labels, and so on?
  - How can I graphically draw them?
    - Example, 'ConnectorZ' is a two pixel-wide dashed line with an open arrow head
- File extension is gmfgraph
- You can re-use gmfgraph files for different editors
- Rational Software Architect includes a wizard to automatically create a default gmfgraph file based on an EMF Ecore file
- A gmfgraph file is NOT linked to any specific ecore file

17

# Tooling Def Model (gmftool)

- Defines the tools for the custom editor
  - Palette and menu entries
- File extension is gmftool
- Generally, tooling definition is domain model-specific, and not appropriate to re-use between different custom diagrams
- Rational Software Architect includes a wizard to automatically create a default gmftool file based on an EMF Ecore file
  - Can be created or extended by hand
- A gmftool file is NOT linked to any specific ecore file



# Mapping Model (gmfmap)

- Ties together (maps) the graphical definition (gmfgraph), tooling definition (gmftool) and domain model (ecore)
- File extension is gmfmap
- Created using a wizard
  - Can be extended and refined by hand
- For example: link together a domain model node with its graphical definition and its tools (palettes and menus)

19

# Diagram Gen Model (gmfgen)

- Defines the custom editor's code generation options
  - For example, property defines if Print support should be included
- File extension is gmfgen



## Generated Code

- From the Diagram Gen Model, you generate the custom diagram's code
- Generates a new Eclipse plugin project
  - Fully-configured plug-in
  - Includes the generated Java source code
- Additional customization and enhancement can be made to the generated editor
  - Edit the generated code
  - It is designed to be extensible, and can be extended with additional plug-ins

21

## GMF and Rational Software Architect Models

- GMF Diagrams can be stored outside of Rational Software Architect Model files
- GMF Diagrams can be stored inside of Rational Software Architect Model files
- GMF Diagrams can reference, display, and manipulate UML information
- So, GMF can be used to extend Rational Software Architect Model capabilities

22

# Lab 2: Build Console GMF Example

- Given:
  - The ongoing Console Transformation example and the generated EMF wrappers for its input files
- After completing this lab, you will be able to:
  - Create and run a GMF generated graphical editor

23

# Further Information

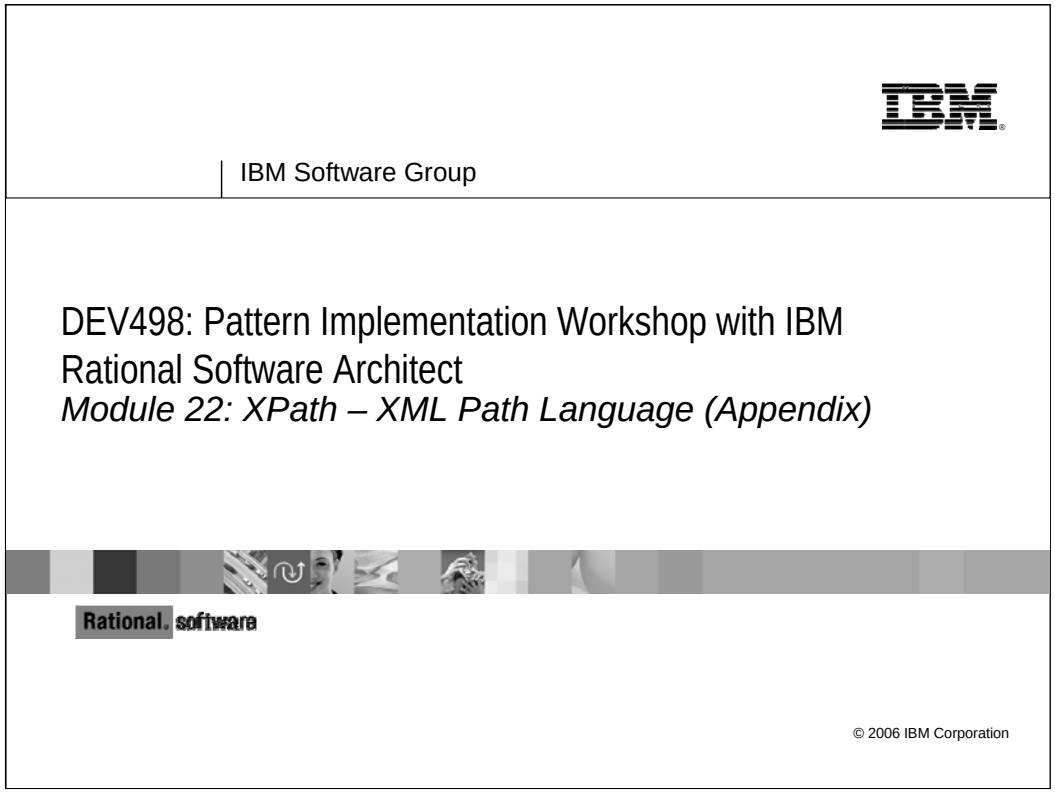
- Web resources
  - www.eclipse.org/gmf
    - Eclipse page for GMF
  - www.eclipse.org/emf
    - Eclipse page for EMF
  - www.eclipse.org/gef
    - Eclipse page for GEF


24

IBM

**Web Resources**

- www.eclipse.org/gmf
  (Eclipse page for GMF)
- www.eclipse.org/emf
  (Eclipse page for EMF)
- www.eclipse.org/gef
  (Eclipse page for GEF)




## Contents

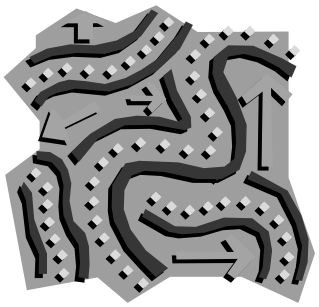
# XPath – XML Path Language

- Objectives:
  - Describe the reasons for using XPath
  - Define the components and constructs that make up the XML Path Language
  - Describe how XPath can reference data in XML documents
  - Write simple XPath expressions
  - Identify abbreviated XPath expressions
  - Describe how to partition the XPath document

2

## What is XPath?

- XPath is a specification for querying an XML document.
  - Originally designed for use by XSLT and XPointer.
  - Now used by many XML-related technologies, such as XQuery.
- XPath satisfies the need to address (locate) parts of a document which meet specified criteria.
  - Example: In the XML description of a book, "find all chapters with 'Java' in the title."
- XPath provides the ability to address any slice of an XML document in any direction.
- XPath is a W3C Recommendation.
  - November 16, 1999



3

IBM

XPath was defined during the development of XSLT (XML Stylesheet Language Transformation) and XPointer. It was designed to provide unambiguous traversal of XML documents.

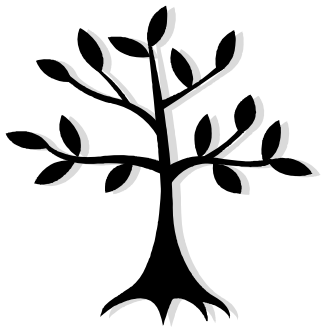
XPointer and XSLT both use XPath's functionality, but XSLT uses only a subset of XPath, while XPointer uses additional syntax to extend its functionality.

XPointer allows forward and backward addressing to specific XML locations internal to a document, and to locations in external XML documents.

XQuery is an emerging technology that will eventually provide standardized access to RDBMS data stores that use XML.

## Why is it Called XPath?

- XML documents are frequently viewed as a tree of nodes.
- Expressions describe a path to a given node or set of nodes (*node-set*).
- Consider the DOS, UNIX, or URI syntax for addressing files in a directory structure.
  - `/publications/articles/Transformations.xml`
  - This is called a *pathname* to the file.
  - It describes the *path* to follow, from the root, through a tree of directories (folders), to locate a given file.
- Similarly, XPath also uses a forward slash to separate the nodes of a path.



4

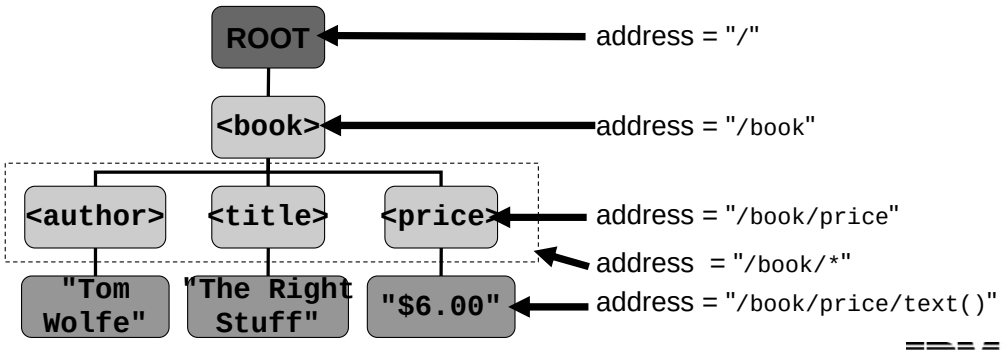
IBM

Paths are a natural way to express a hierarchical structure.

DOS and Windows actually use a backslash to represent the path separators. URI's, XPath, and most other path addressing schemes use a forward slash, as backslash is used to express or escape special characters. For example, '\t' represents a TAB character, and '\\' represents a backslash.

## Example Tree Representation of XML

```
<?xml version="1.0"?>
<book>
  <author>Tom Wolfe</author>
  <title>The Right Stuff</title>
  <price>$6.00</price>
</book>
```




5

This example shows a typical XML document and how it is represented as a tree of nodes. There is a single root node, that contains several other types of nodes.

There are seven node types in XML. They are:

1. root nodes
2. element nodes
3. text nodes
4. attribute nodes
5. namespace nodes
6. processing instruction nodes
7. comment nodes

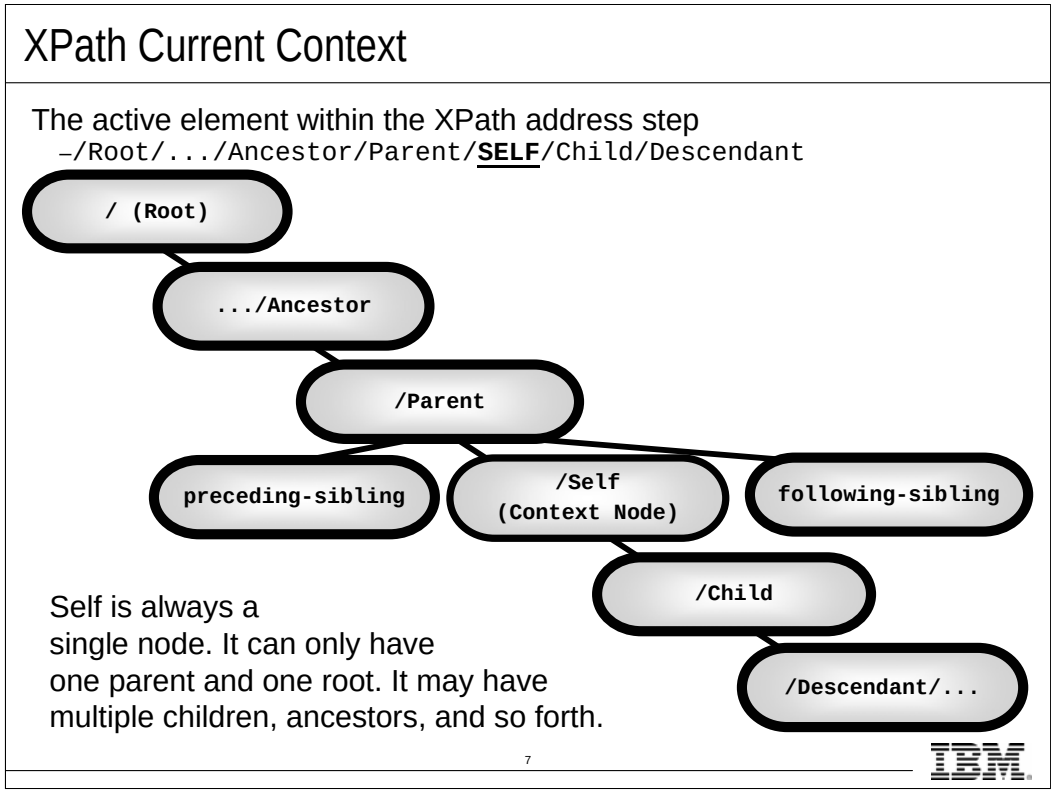
## XPath Expression Evaluation

- An XPath expression is a series of *steps*.
  - A step is a search criteria statement.
  - Example: "find figures in the current chapter."
- An XPath expression has a *current context*.
  - A node in the tree that is the starting point for the step.
  - Example: "current chapter in the book."
- Each step, except the last, must evaluate to a set of nodes in the XML tree.
  - Example: "all the chapters in a book."
  - Steps are evaluated against one or more nodes.
  - The resulting set of nodes may be empty.
- The last step returns one of these:
  - Number
  - Boolean
  - String
  - Node-set

6

Think of an XPath expression as a series of steps through the XML tree. Each step is a rung in the ladder, or layer of the tree.

Wildcards permit a single step to represent many layers, much like skipping several rungs when climbing down the ladder.




The current context is simply a "you are here" designation within a complete XPath address.

As an XPath expression is evaluated, the current context usually shifts.

Relative paths do not make sense as standalone entities. They must be combined in some other context based on the document root.

## XPath Step Syntax

- An XPath location path is made up of one or more steps separated by forward slashes ("/").
- Each step within the path consists of:
  - **Axis**: Branch of the node tree relative to the current context node.
  - **NodeTest**: Tests node for inclusion.
  - **Predicate**: Optional filter of matched nodes.
- Example:
  - Locate all chapters titles in the book that contain the string 'XPath':

```
/book/child::chapter/child::title[contains(text(),'XPath')]/
```

```
.../axis::nodeTest[predicate]/...
```

8

XPath provides a simple method to traverse an XML tree structure, and to select a slice of information in any direction as defined by the Axis.

Paths starting with a forward slash are absolute paths from the root downward through the document tree; paths not beginning with a slash are relative to the current (context) node of the node list.

XPath is not a language, but more of an addressing syntax used to identify slices of information within an XML document. XPath uses a path notation to define locations within a document. For brevity, this syntax does not use XML constructs.

XPaths, when expressed in an XML document, usually appear as an attribute value, as in an xsl:template element in XSLT.

## XPath Address Notation

- An *address* is a node (or several nodes) in a tree that is your starting point for searching.
- Abbreviated syntax is allowed for several different axes.
  - "`child::`" has an empty default as it is the default axis
    - Example: "`/child::catalog/child::tools/`" is the same as "`/catalog/tools/`"
- A complete XPath expression may consist of only a location path.
- Absolute location path:
  - Starts search at the root of the tree
  - Search begins with a forward slash
    - Example: `/catalog/tools`
- Relative location path:
  - Sequence of one or more location steps, or referenced from the current context node.
    - Example: `catalog/tools`

9

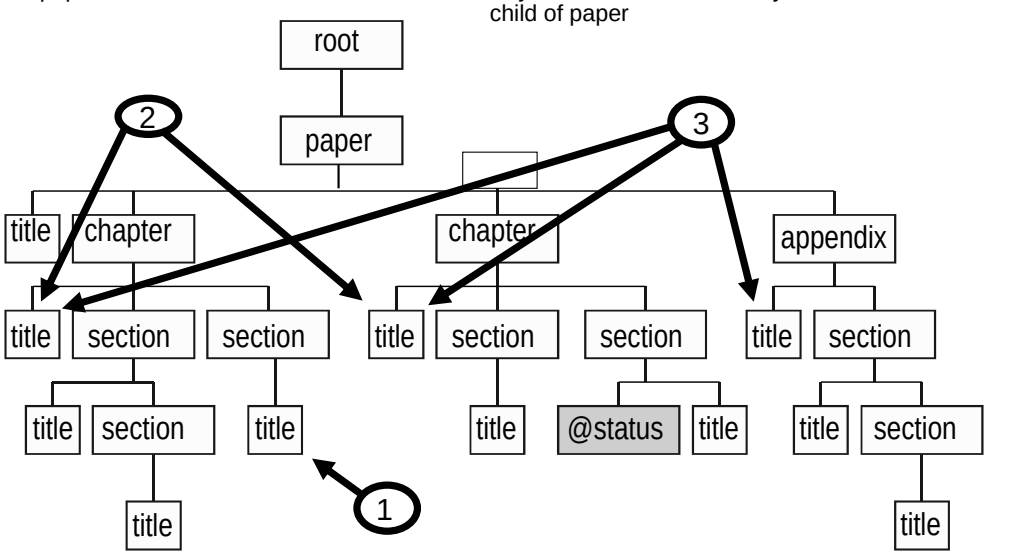
All axes and abbreviations will be discussed later in this unit.

Absolute paths are sometimes called fully-referenced or full paths.

Relative paths are sometimes called partial paths.

# Example: Absolute Addressing

- /paper/chapter[1]/section[2]/title — Title for first chapter, second section
- /paper/chapter/title — Titles for all chapters
- /paper/*/title — Any title that is a child of any child of paper

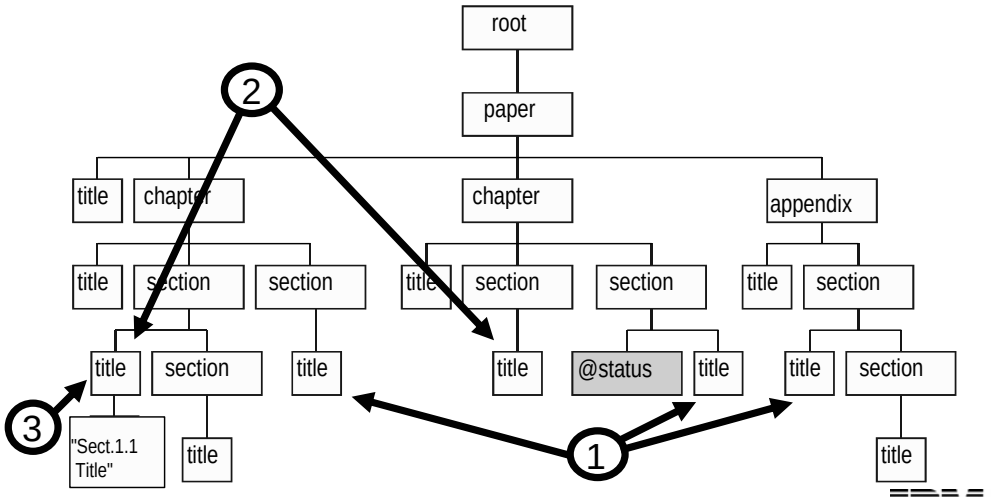



10

# Example: Absolute Addressing with Predicates

**Instructor Notes:**

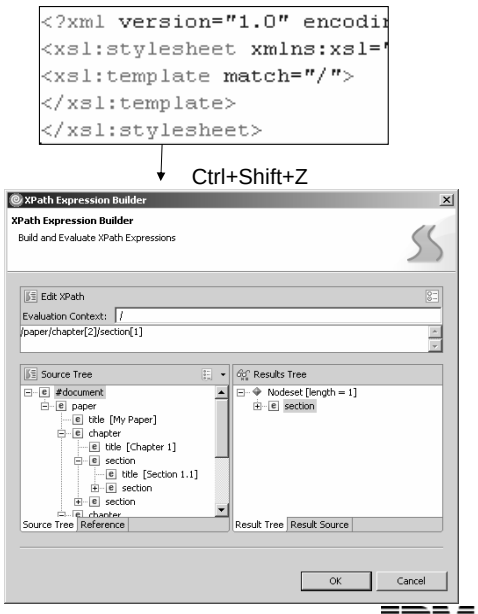
1. /paper/*/section[last()]/title — Titles for last sections
2. /paper/*/section[last()-1]/title — Titles for the second-to-last sections
3. /paper/chapter[1]/section[title='Sect.1.1 Title']/title — Select title by name




11

## Testing XPaths in Rational Application Developer

- Rational Application Developer provides the XPath Expression Builder to build and evaluate XPath Expressions.
  - Available from within XSL editor.
  - To use, position cursor within an **<xsl:template>** tag and choose **XPath Expression...** from the context menu, or press **Ctrl+Shift+Z**.
- Allows expressions to be built "by example" from elements in a representative document, or entered by hand.
- Results are shown both as a tree and in terms of source.

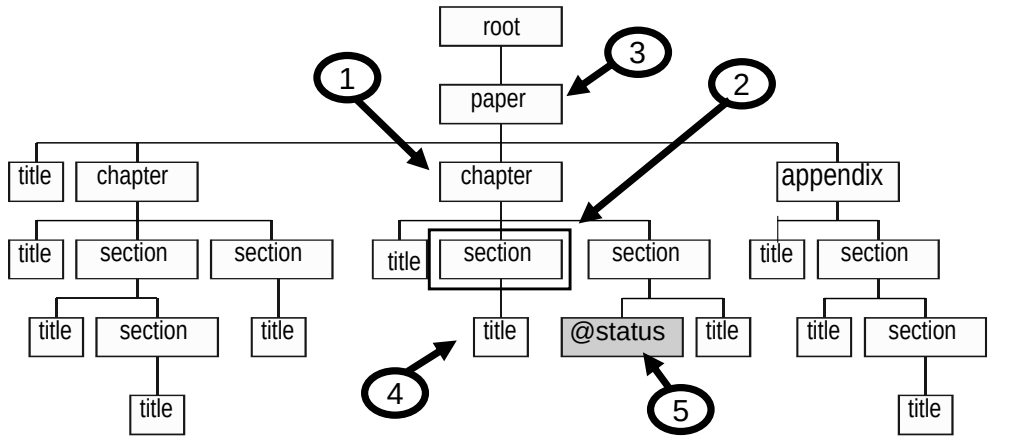



12

## Example: Relative Addressing

**/paper/chapter[2]/section[1] - Absolute path to "current context"**

| | |
|---|---|
| 1. `parent::node()` *or* `..` | Parent of current context |
| 2. `self::node()` *or* `.` | Context node (self) |
| 3. `../..` | Parent of parent of context node |
| 4. `child::*` (default) | Children of the current context node |
| 5. `./following-sibling::node()/@status` *or* `./following-sibling::*/@status` | Status attribute of any following sibling node siblings1 |




13

IBM

This chart demonstrates relative addressing, based on a current context of the first section of the second chapter. It introduces the abbreviations "." (for self, the current node) and ".." (for the parent of self). These abbreviations are similar to those used in Windows and UNIX® file systems.

The chart also introduces the @ notation for identifying attributes.

The examples show the following: The parent of the current context is the second chapter element.

1. The current context.
2. The parent of the parent of the current context.
3. The default next step of the current context is always its child or children.
4. The status attribute of the sibling element that follows the current context (in this case, section) .

## XPath: The Thirteen Axes

| Axis Name | Description |
|---|---|
| **1. ancestor** | Ancestors of context node: parent, grandparent, and so on |
| **2. ancestor-or-self** | Context node and its ancestors |
| **3. attribute** | Attributes of the context node |
| **4. child** | Children of the context node |
| **5. descendant** | Descendants of the context node: child, grandchild, and so on |
| **6. descendant-or-self** | Context node and its descendants |
| **7. following** | All nodes that follow the context node, not including descendants, attributes, and namespaces |
| **8. following-sibling** | All siblings that follow the context node |
| **9. namespace** | Namespace node of context node |
| **10.parent** | Parent of context node if it exists. Parent of attribute or namespace is the element that contains it. |
| **11.preceding** | All nodes that are before the context node, not including ancestors, attributes and namespaces |
| **12.preceding-sibling** | All siblings that precede the context node |
| **13.self** | The context node |

14

There are 13 axes defined in XPath that enable you to search different parts of the XML Document from the current context node or the root. Despite the singular form of axis names (such as "ancestor" and "child"), only parent and self always refer to a single node.

All axes can be used in both relative and absolute paths.

## Abbreviated Step Notation

| | Expansion | Example |
|---|---|---|
| | `child::` | **`chapter/section`** expands to **`child::chapter/child::section`** (all the **section** children of all the **chapter** children of the context node) |
| `.` | **`self::node()`** | **`./attribute::name`** expands to **`self::node()/attribute::name`** (the **name** attribute of the context node) |
| `..` | **`parent::node()`** | **`../attribute::name`** expands to **`parent::node()/attribute::name`** (the **name** attribute of the parent of the context node) |
| `@` | **`attribute::`** | **`./@name`** expands to **`self::node()/attribute::name`** (the **name** attribute of the context node) |
| `//` | **`/descendent-or-self::node()/`** | **`.//chapter`** expands to **`./descendant-or-self::node()/chapter`** (all the **chapter** descendants of the context node) |

15

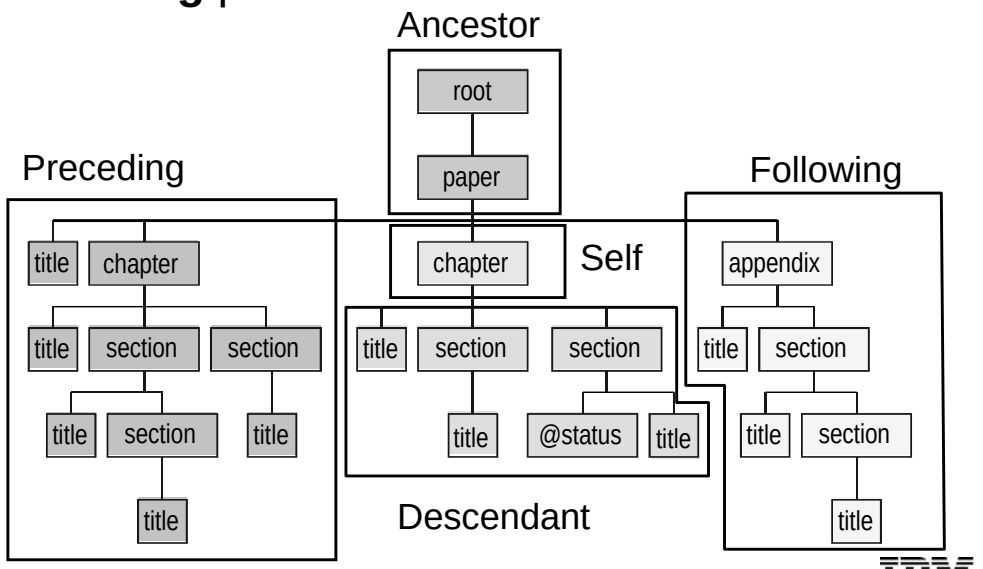
The commonly-used axes, such as attribute, child, and descendent-or-self, have a shorthand syntax.

If the shorthand syntax is used the "::" separator that follows the axis name is omitted.

`child::` is the default axis if no axis is specified.

## XPath: Partitioning the Document

- **Self**, **ancestor**, **descendant**, **preceding**, and **following** partition the entire document.

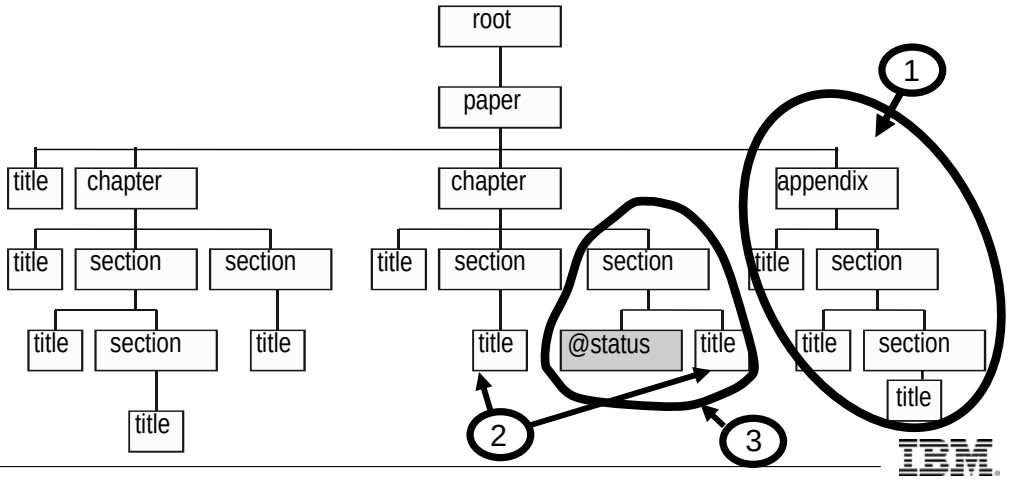



For the node labeled "Self" (the current context node), the labels on the various nodes indicate their axis relationship to "Self".

These five axes contain all the nodes within the document, and do not overlap.

## Example: Addressing with Axes

1. /paper/chapter[last()]/following::* — Everything after the last chapter.
2. /paper/chapter[2]/descendant::node()/title — All title descendants of chapter 2.
3. //*[attribute::status] or //*[@status] — All element nodes containing a status attribute.




These samples depict the variety and scale of simple XPath queries using different axis notation. This path extracts everything following the last chapter in the book.

1. This path extracts every title element up to and including the second chapter.
2. This path extracts all the descendents of self that have a status attribute.

## XPath Axis Node Type and Node Tests

| Axis type | Type of nodes returned |
|---|---|
| attribute | attribute node |
| namespace | namespace node |
| all other axes | element node |

| Node test | Result |
|---|---|
| `* (Wildcard)` | Select all nodes of the given axis type. |
| `QName` | Selects node if it has the specified namespace qualified name (if namespace is null, then name is not in any namespace). |
| `NCName:*` | Selects node if it has the specified namespace. |
| `text()` | Returns the node's body text. |
| `processing-instruction()` | Returns the processing instruction (for PI nodes). The processing-instruction node test can have an optional predicate which contains a literal. |
| `comment()` | Returns the comment (for comment nodes). |
| `node()` | Is true for any node of any type whatsoever. |
| `id("value")` | Returns the node containing an ID type attribute of the specified value. |

18

The first table lists the types or axes, and the corresponding type of node returned. This list only indicates the principal node type. For example, an axis of child::* will return nodes of type element, but the returned elements may have child nodes that are of type attribute.

The second table lists the node tests and the resulting node (or node list). A node test follows the Axis in the address step, and qualifies the nodes to be included or excluded in the search. The most common form of node test is the QName or actual element name.

The wildcard ("*") node test selects all nodes of the given type. For example, `attribute::*` selects all attributes.

## Sample Node Tests

`//comment()`

- Extract all comments from a document.

`/book/*/title`

- Extract all top-level titles regardless of parent type (that is, Chapter, Appendix, and so on).

`/processing-instruction()`

- Extract all processing instructions that exist outside of the root element.

`/book/chapter[2]//text()`

- Extract the actual text from all elements inside the second chapter.

`chapter/section[2][@status="Draft"]`

- Extract the second section child of every chapter child of the context node where the section status attribute has a value of "Draft".

19

## Predicates

- All comparisons or function calls are within the predicate, enclosed within [ ].
- Predicates test a set of nodes and return one of:
  - A new set of nodes
  - A string
  - A Boolean
  - A number
- Each node in the list of nodes is tested to see if the predicate is true.
  - If predicate is true then the node is included in the resulting list of nodes.
- If a predicate results in no matching nodes, an empty result set is returned.

20

Predicates filter a list of nodes. Predicate expressions can be function calls, numbers, literals, or location paths.

## Predicate Expressions

- Predicate expression types:
  - Function call
  - Number
  - Literal
  - Location path
- Operators may be used inside a predicate.
  - Node-set
    `|` (union)
  - Boolean
    `and or`
  - Relational
    `= != < > <= >=`
  - Arithmetic
    `+ - * div mod`

21

IBM

Predicates offer a wide variety of built-in functions to aid in filtering nodes. A predicate may consist of a single test, which may itself consist of a direct address node index, or a boolean function. However, most predicate tests consist of one or more comparison operations. Multiple tests can be combined within a single predicate test using operators.

A predicate may combine two node-sets using the union ("|") operator.

A predicate expression may contain logical operators. If A and B are expressions with a boolean value (such as "a=1"), then A and B is true if both expressions are true, and A or B is true if either condition is true. There is no "not" operator, but the `not()` function (described later in this unit) may be used instead.

The div operator performs floating point division. The mod operator provides a remainder function.

## Predicate Core Functions

| Function | Returns | Description |
|---|---|---|
| `last()` | number | Returns the index of the last node in the current context, that is, the context size. |
| `position()` | number | Returns the index of the current node within the context. |
| `count(node-set-expr)` | number | Returns the number of nodes in the node-set identified by the given expression. |
| `id(object)` | node-set | Returns a node-set containing the nodes that have the specified IDs. |
| `local-name (node-set-expr)` | string | Split a fully qualified name (namespace:object) and return the object's name. |
| `namespace-uri (node-set-expr)` | string | Split a fully qualified name and return the namespace URI |
| `name(node-set-expr)` | string | Returns the fully qualified name for the first node in the node-set |

22

The table lists the XPath predicate functions that are part of the core function library. A node-set-expr is a relative or absolute path.

For the id function, the object parameter may contain more that one node, in which case the returned node-set may contain more than one node.

A few functions, such as local-name and Namespace-URI, have optional arguments. If no argument is present, the current context node is treated as the argument.

Examples:

`/child::chapter[position()=1]` returns the first chapter element that is under the document root.

`/chapter[1]` is the abbreviated form of the same expression.

## Predicate String Functions (1 of 2)

| String Functions | Return Type | Description |
|---|---|---|
| `string (object)` | string | Converts **object** into a string. |
| `starts-with (source, target)` | Boolean | Returns **true** if **source** starts with the characters of **target.** |
| `contains (source, target)` | Boolean | Returns **true** if **source** contains the characters of the **target**. |
| `substring-after (source, target)` | string | Returns the substring of **source** following the first occurrence of **target**. |
| `substring-before (source, target)` | string | Returns the substring of **source** preceding the first occurrence of **target**. |
| `substring (source, index, count)` | string | Returns a substring of **source**, starting at **index** for an optional **count**. |

23

Almost any object type can be passed into string functions. The processor will attempt to convert non-string objects to their string representation. Booleans are converted to the strings "true" and "false". The string value of an element is the concatenation of all the characters of the element and its descendants.

In the string function, only the first node of the argument node-set is converted to a string.

## Predicate String Functions (2 of 2)

| String Functions | Return Type | Description |
|---|---|---|
| `string-length (string)` | number | Returns the **string** length. |
| `concat (string, string, ...)` | string | Returns a concatenation of its arguments. Must have at least two arguments. |
| `normalize-space (string)` | string | Removes leading and trailing whitespace, and replaces adjacent whitespace characters with a single whitespace. |
| `translate (source, from, to)` | string | Returns **source** with each character that appears in **from** replaced by the corresponding character in **to**. |

24

## Predicate Number and Boolean Functions

| Number Functions | Description |
|---|---|
| `number (object)` | Converts an object to a number. |
| `sum (node-set)` | Returns the sum of values of nodes of the node set. |
| `floor (number)` | Returns the largest integer that is not greater than argument (rounds down). |
| `ceiling (number)` | Returns the smallest integer that is not less than argument (rounds up). |
| `round (number)` | Returns the closest integer to argument. |

| Boolean Functions | Description |
|---|---|
| `not (boolean)` | Returns **true** if the argument is **false** and **false** otherwise. |
| `true ( )` | Returns **true.** |
| `false ( )` | Returns **false.** |

25

The number functions all return numbers. The boolean functions all return booleans.

The `number()` function attempts to convert its argument or the current context to a number. If it is unable to do this, it returns NaN ("Not a Number").

## XPath 2.0

- XPath 2.0 is more powerful and more complex than XPath 1.0.
- XPath 2.0 processes *sequences*.
  - Like a node-set, but can include additional atomic values
  - Ordered set of values without duplicates
- XPath 2.0 replaces the primitive XPath 1.0 data types with XML Schema data types.
  - For example, XPath 1.0 has no date-time data types.
- Additional functions augment the XPath 1.0 ones
- XPath 2.0 is a syntactic subset of XQuery 1.0.

26

XPath 2.0 became a W3C candidate recommendation in June 2006. For the basic specification, see http://www.w3.org/TR/xpath20/. For the specification pertaining to the new XPath 2.0 functions, see http://www.w3.org/TR/xpath-functions.

## Checkpoint Questions (1 of 3)

1. Which of the following items are part of the XPath step syntax?
   a. Predicate
   b. AxisName
   c. Ancestor
   d. Ceiling
   e. NodeTest

2. The axis shorthand notation of // indicates what?
   a. Ancestor
   b. Parent
   c. Ancestor-or-self
   d. Descendant-or-self

27

## Checkpoint Questions (2 of 3)

3. Which XPath statement will return the number of questions on a test?
   a. `count(/test/question)`
   b. `/test/question/count()`
   c. `/test[count(question)]`
   d. None of the above

4. The predicate function starts-with ("XML is Great", "XML") will return:
   a. XML
   b. true
   c. Is Great
   d. False
   e. XML is Great

28

## Checkpoint Questions (3 of 3)

5. What will be the results of the following XPath:

```
/news/story[@year='2001']/self::node()[cont
ains(text(),'IBM')]/
```

a. All 2001 news stories that contain IBM inside the text element
b. All news stories with a year element = 2001 and a text element of IBM
c. Any news story with either IBM or 2001 in its text
d. All 2001 news stories that contain the letters IBM in any order
e. Error, as this is an invalid XPath statement

29

## Further Information

- Rational Software Architect Help Topics
- Web Resources


30

IBM

**Rational Software Architect Help Topics**

- Developing Applications and Websites > Building XML applications > Creating XPath Expressions

**Web Resources**

- W3C XPath specification

  www.w3.org/TR/xpath
- Interactive tutorial

  http://www.zvon.org/xxl/XPathTutorial/General/examples.html
- Expression testers

  http://www.zvon.org:9001/saxon/cgi-bin/XLab/XML/xlabIndex.html?stylesheetFile=XSLT/xlabIndex.xslt
- Axis Powers (two parts)

  http://www.xml.com/pub/a/2000/12/20/xpathaxes.html

  http://www.xml.com/pub/a/2001/01/03/xpathaxes.html
- Finding Relatives

  http://www.xml.com/pub/a/2000/10/04/transforming/trxml5.html